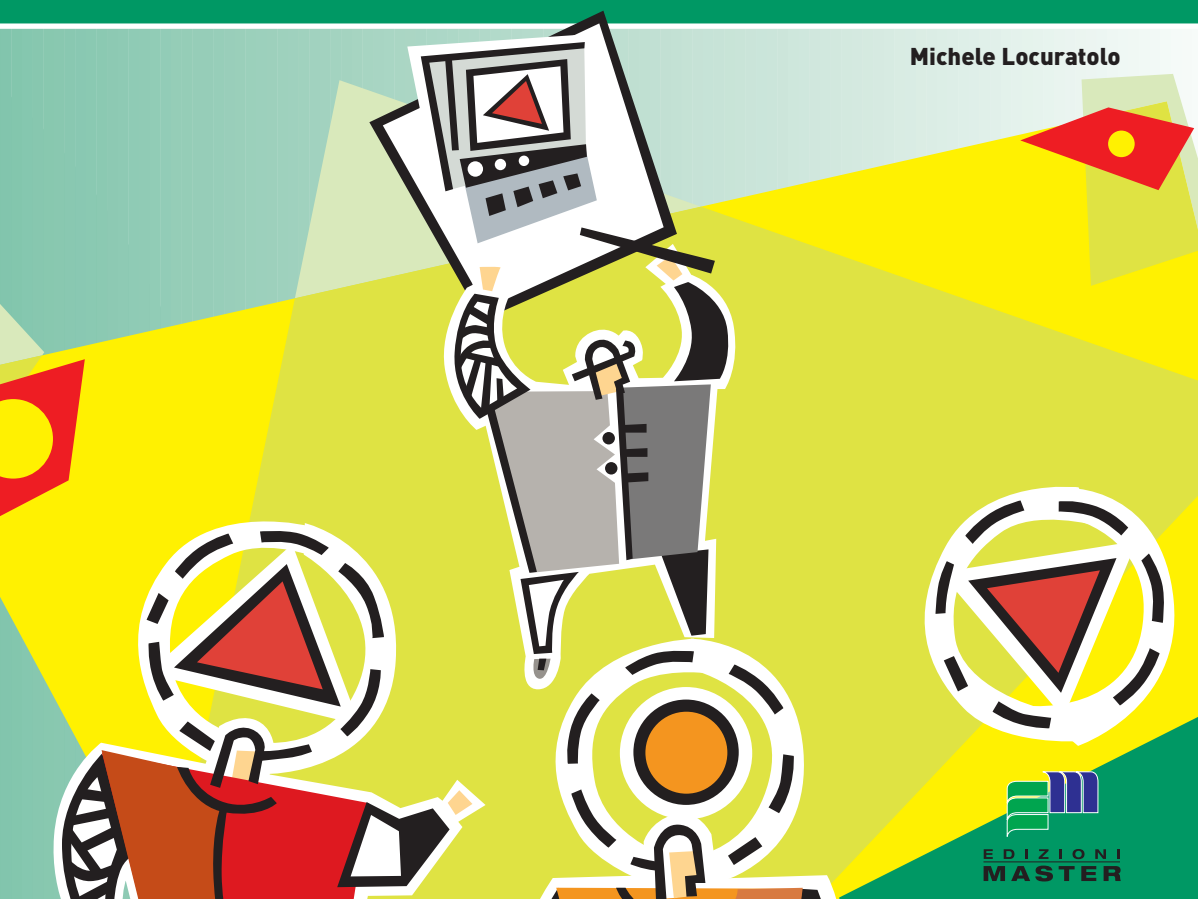
IL LINGUAGGIO PIÙ ELEGANTE. PER I PROGRAMMATORI PROFESSIONISTI E PER CHI ASPIRA A DIVENTARLO

IMPARARE C#

Michele Locuratolo

EDIZIONI MASTER

i libri di

IMPARARE C#

di Michele Locuratolo

INDICE

Array indici e collections

Novità C# 2.0

PuntoInformatico | Libri

INTRODUZIONE

Era l'anno 2001, quando iniziai ad avvicinarmi al .NET Framework. Questa nuova tecnologia era ancora in versione beta con tutto ciò che ne conseguiva. Da allora sono passati cinque anni ed il .NET Framework studiato, realizzato e distribuito dalla Microsoft ha preso ormai piede in diversi ambiti e settori, a dimostrazione che la direzione presa cinque anni fa era probabilmente la più giusta.
Nel corso di questi anni, il .NET Framework è cresciuto e maturato fino ad arrivare alla versione 2.0 (o 2005) corredato da svariati tool di sviluppo estremamente potenti e versatili. Indubbiamente, il più potente editor con cui scrivere software in .NET è Microsoft Visual Studio 2005. a cui si affiancano delle versioni più leggere denominate Express.
Per scrivere software in CSharp 2005, la versione da scaricare gratuitamente è Microsoft Visual CSharp 2005 Express Edition, scaricabile dal seguente indirizzo:
http://msdn.microsoft.com/vstudio/express/visualcsharp/.
Il .NET Framework è una tecnologia abbastanza vasta che abbraccia diversi aspetti legati alla programmazione e diversi linguaggi (ad oggi sono circa 42 quelli supportati). Si va dalla realizzazione di software per Windows alla realizzazione di programmi per dispositivi mobili come computer palmari e smartphone.
Tra questi due estremi si collocano i servizi web, le applicazioni web ed i servizi Windows. In questo libro, ci concentreremo principalmente su uno dei linguaggi supportati da questa tecnologia: C# 2005.
Dato questo è un linguaggio orientato agli oggetti, nei capitoli 4 e 5 saranno analizzati i principi di base di questa eccezionale metodologia di programmazione cercando di focalizzare la nostra attenzione su quelli più importanti.
Parleremo inoltre dei potenti generics, dei comodi iterators e delle altre novità molto utili ed interessanti, tutte introdotte per rendere più semplice e veloce il nostro lavoro quotidiano, consentendoci di svi-

luppare applicazioni sempre più potenti e veloci. Più o meno tutti i paragrafi, ad eccezione di quelli prettamente teorici, sono corredati da esempi pratici, semplici e funzionanti i cui sorgenti sono disponibili in allegato alla rivista.
Con la speranza che questo libro serva da introduzione a questa potente tecnologia e convinca i più restii a fare il "grande salto", auguro a tutti una buona lettura.

IMPARIAMO C#2005

Il linguaggio CSharp, sin dalla prima versione, viene eseguito in un ambiente denominato Microsoft .NET Framework. In questo capitolo vedremo alcuni degli aspetti base di questa piattaforma per comprendere meglio tanto il linguaggio quanto l'ambiente di esecuzione.Capire questo aspetto è molto importante ai fini della reale comprensione delle potenzialità di questa tecnologia e dei suoi ambiti di esecuzione. I concetti spiegati in questo capitolo, ad una prima analisi, potrebbero apparire astratti e poco attinenti con il tema di questo libro. Per questa ragione si è cercato quanto più possibile di non scendere nei dettagli implementativi del Framework.
Si consiglia comunque una lettura della abbondante documentazione on line per approfondire gli argomenti che più stuzzicano la curiosità del lettore.Iniziamo quindi dai mattoni che compongono il .NET Framework.

1.1 IL .NET FRAMEWORK

Con il termine .NET Framework si indica l'ambiente, creato da Microsoft, con cui è possibile creare ed eseguire applicazioni scritte con uno dei linguaggi compatibili.Il .NET Framework si compone da diversi mattoni tra cui *CLI* (Common Language Infrastructure) che "descrive" il codice eseguibile, *BCL* (Base Class Library) che raccoglie un vasto insieme di classi riusabili, *CLR* (Common Language Runtime) che gestisce l'esecuzione del codice e *CLS* (Common Language Specification).
Vediamo in breve di cosa si tratta.

1.1.1 CLI (Common Language Infrastructure)

La Common Language Infrastructure (CLI) è una specifica e standardizzata sviluppata da Microsoft, che descrive il codice eseguibile e l'ambiente di esecuzione dei programmi scritti in .NET. CLI rap-

presenta uno dei mattoni di base del .NET Framework. La specifica definisce un ambiente che permette a più linguaggi di alto livello di operare assieme per la realizzazione di software e di essere utilizzati su piattaforme diverse senza la necessità di essere riscritti per specifiche architetture. La specifica Common Language Infrastructure, a sua volta, si compone di 4 mattoni fondamentali:

- Il CTS (Common Type System) che costituisce l'insieme dei tipi di dato e di operazioni possibili su di essi (vedi Cap. 3, par. 1).
- I Metadati che rappresentano le informazioni sulla struttura del programma. Essi sono indipendenti dal linguaggio di partenza per permettere a programmi scritti in linguaggi differenti di comunicare tra loro.
- La CLS (Common Language Specification) che definiscono le specifiche di base che un linguaggio deve implementare per essere compatibile con .NET Framework.
- Il VES (Virtual Execution System) che rappresenta il sistema che esegue i programmi CLI.

Spesso, quando si fa riferimento al codice scritto in uno dei linguaggi supportati dal .NET Framework, si usa il termine managed. Tale terminologia è riferita appunto a come il Virtual Execution System (VES) gestisce l'esecuzione del codice. Tutto il codice scritto in .NET è definito managed. Codice scritto in altri linguaggi (come C o C++), sebbene sia usabile anche dal .NET Framework attraverso un meccanismo denominato Platform Invocation (PInvoke) è definito, per ovvie ragioni, unmanaged.

1.1.2 BCL (Base Class Library)

Un altro mattone che compone il .NET Framework è la Base Class Library (BCL) che raccoglie un insieme di classi riusabili al fine di comporre le nostre applicazioni scritte in .NET.

La BCL è organizzata in Namespaces (spazi di nomi) asseconda della funzionalità specifica delle rispettive classi. Il Namespace principale si chiama System. Al di sotto di esso troviamo, ad esempio, System.Windows.Forms, System.Web, System.Data. Tali Namespace descrivono abbastanza bene il loro ambito di applicazione. Ci occuperemo più dettagliatamente dei namespace, del loro utilizzo e della loro creazione nel capitolo 6. Le classi della BCL sono talmente tante che solo con l'uso continuato del .NET Framework se ne prende dimestichezza. Un utile riferimento a tutte le classi della Base Class Library è disponibile in Visual Studio 2005 nel tool denominato "Object Browser". Per accedere alla visualizzazione della BCL è sufficiente selezionare il menù view e la voce object browser.

1.1.3 CLR (Common Language Runtime)

Il CLR è il componente del .NET Framework che gestisce a tutti gli effetti l'esecuzione di un programma. Sfruttando i mattoni elencati nei precedenti paragrafi, il CLR permette a componenti scritti da linguaggi diversi di comunicare tra loro. CLR è anche denominato "Ambiente di gestione dell'esecuzione".
In pratica, un programma scritto in .NET, attraverso il CLR, viene eseguito nel Virtual Execution System (una sorta di macchina virtuale) ed è quindi svincolato dall'hardware del PC su cui è in esecuzione.
Quando un programma, scritto in uno qualsiasi dei linguaggi supportati dal .NET Framework, viene compilato, il risultato della compilazione è un componente composto da una forma intermedia di codice chiamata Microsoft Intermediate Language (MSIL) e da una serie di metadati che indicano al CLR alcune informazioni come la versione, il linguaggio utilizzato etc.Al momento dell'esecuzione vera e propria, ciascun componente per il quale è stato generato codice MSIL viene compilato in modalità JIT (Just-In-Time) alla prima chiamata. La compilazione JIT traduce il linguaggio MSIL in codice nativo quindi comprensibile dal processore. Alla successiva esecuzione dello stesso componente, viene eseguito direttamente il codice nativo già

compilato in precedenza.
Questo processo di compilazione in modalità JIT e di esecuzione del codice viene ripetuto fino a completamento dell'esecuzione.
Ad una prima analisi, l'esigenza di compilare due volte un'applicazione (la prima per tradurre il codice scritto in MSIL e metadati e la seconda per l'esecuzione vera e propria del programma), potrebbe sembrare controproducente. Un programma scritto in C++ ad esempio, viene compilato una sola volta e, al momento dell'esecuzione, è direttamente eseguito.
Il vantaggio di eseguire il programma direttamente però, introduce uno svantaggio: la necessità di ricompilare il programma se deve essere eseguito su un processore diverso.
Svantaggio eliminato dal .NET Framework attraverso la compilazione JIT.
Grazie ad essa infatti, il componente che conosce l'hardware installato sulla macchina di esecuzione non è più il nostro software ma il Framework.
Ne consegue che il nostro componete scritto in .NET, sarà eseguito senza problemi su qualsiasi hardware, senza la necessità di crearne versioni specifiche.

1.1.4 Metadati

I metadati rappresentano le informazioni relative alla struttura del programma.
Sono generati automaticamente dal CLR nel momento in cui viene effettuata la compilazione.
Questo è il processo con cui, il nostro codice scritto in uno dei linguaggi supportati dal .NET Framework, viene convertito nel linguaggio MSIL utilizzato poi dal JITter. I metadati sono memorizzati all'interno dell'assembly ed includono informazioni relative a:

- Assembly: in questo contesto, i metadati sono utili a descrivere alcune caratteristiche dell'assembly stesso come la lin-

gua, la versione, i tipi usati e quelli referenziati e le impostazioni di sicurezza dell'assembly stesso.
- Tipi: queste informazioni sono relative ai tipi e comprendono il nome del tipo, le eventuali classi base ed interfacce, metodi, campi etc.
- Attributi: sono relativi agli attributi definiti dall'utente o a quelli usati direttamente dal .NET Framework.

Tutte queste informazioni, fanno sì che il nostro componente possa, in un certo senso, autodescriversi, rendendo possibile tanto l'interoperabilità tra i vari linguaggi, quanto una più efficiente compilazione Just In Time.

1.2 IL LINGUAGGIO CSHARP

Dopo aver appreso quali sono gli elementi che compongono il .NET Framework, vediamo dove si colloca il linguaggio CSharp.
Esso è semplicemente uno dei diversi linguaggi che si attengono alle specifiche del CLS (Common Language Specification) e che quindi vengono "compresi" dall'ambiente di esecuzione e possono essere eseguiti dal CLR. Il fatto di essere CLS-Compliant, come abbiamo visto in precedenza, oltre a permettere al programma di essere eseguito dal CLR, consente a programmi scritti in CSharp di comunicare con altri programmi scritti in un altro linguaggio compatibile nonché di accedere alla BCL.
Uno dei tanti punti di forza di questa tecnologia è proprio l'interoperabilità tra i linguaggi.Prima dell'arrivo del .NET Framework, l'interoperabilità era comunque possibile ma limitata a certi contesti e non semplice da implementare.
Il fatto che ogni linguaggio CLS-Compliant sia tradotto in MSIL in fase di compilazione, consente di usare elementi scritti in diversi linguaggi con estrema semplicità.
Questa caratteristica va a tutto vantaggio della produttività.

E’ possibile ad esempio sfruttare un componente scritto in Visual Basic .NET (o Visual Basic 2005) magari per un'altra applicazione, o ancora, permettere ai programmatori di lavorare con il linguaggio con cui si trovano più a loro agio ed in cui si sentono più produttivi.
Csharp ed in generale la Common Language Infrastructure, sono stati sottoposti a due enti internazionali che si occupano di standardizzazione: l’ECMA e l’ISO/IEC. Informazioni sul livello di standardizzazione sono liberamente consultabili sui rispettivi siti web delle due organizzazioni.

HELLO WORLD

Dopo aver descritto l'ambiente in cui i programmi scritti in C# vengono eseguiti, e compreso come essi vengono gestiti dal .NET Framework, diamo uno sguardo ad un primo, semplice, programma:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace HelloWorld {
    /// <summary>
    /// Classe di avvio del programma
    /// </summary>
    class Program {
        /// <summary>
        /// Avvio del programma
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args) {
            //Scrittura del testo sulla console
            Console.WriteLine("Hello, world!");
        }
    }
}
```

Nelle prime righe si nota subito l'utilizzo di alcuni elementi della BCL (Base Class Library) e più precisamente System, System.Collections.Generic e System.Text.

La direttiva using che precede i namespaces è utile per poter richiamare dei metodi interni al namespace senza doverne usare il nome completo.

Approfondiremo questo aspetto nel capitolo 6. Facendo riferimento al codice di sopra, senza la direttiva using System, per stampare il testo sulla console avremmo dovuto scrivere:

```
System.Console.WriteLine("Hello, World");
```

Subito dopo le direttive using troviamo la definizione di un nostro namespace chiamato HelloWorld (il nome del programma). Questo ci consentirà, come per i namespace della Base Class Library, di richiamare metodi del nostro programma in forma abbreviata.
Un altro elemento molto utile, in tutti i linguaggi di programmazione, sono i commenti al codice. Il loro scopo è quello di fornire delle indicazioni circa il funzionamento di una classe al fine di rendere più semplice la manutenzione o l'utilizzo della stessa da parte di altri sviluppatori. In C#, per dichiarare un commento su una singola riga, si usa la doppia barra (//), mentre per i commenti su più righe, la tripla barra (///).
Alla riga 9 troviamo la dichiarazione della nostra classe. Una classe è l'elemento che viene usato da C# per la creazione degli oggetti. Approfondiremo il discorso nel capitolo 4.
Alla riga 14 troviamo il metodo static void Main() . Un metodo è l'elemento che definisce il comportamento (o i comportamenti) che ha un oggetto. Approfondiremo anche questo discorso nel capitolo 4.
Infine, alla riga 16, viene richiamato un metodo della classe console il cui scopo è quello di mostrare un testo a video.
Il risultato dell'esecuzione del nostro programma è visibile in Figura 1

Figura 1: Il risultato del nostro semplice programma

2.1 FONDAMENTI

Prima di iniziare a toccare con mano C# 2005 e le sue potenzialità, è necessario chiarire alcuni concetti di base come i tipi, le variabili, le espressioni etc. Mentre nei precedenti paragrafi abbiamo analizzato l'ambiente di esecuzione, in questo capitolo vedremo i mattoni fondamentali del linguaggio. Gli elementi base che ci permetteranno poi di realizzare applicazioni anche complesse.
Anche se vedremo questi aspetti legati al linguaggio CSharp 2005, non dimentichiamoci che essi fanno parte del .NET Framework e si collocano più precisamente nel Common Type System visto nel paragrafo 1.1.1. Se ne deduce che, a parte le differenze puramente sintattiche, quello che vedremo di seguito è comune a tutti i linguaggi supportati. La comprensione degli argomenti trattati in questo capitolo è propedeutica per i capitoli successivi.

2.2 TIPI

Una delle caratteristiche importanti del linguaggio C# e di conseguenza di tutti i linguaggi supportati dal .NET Framework è la "tipizzazione forte".
Questo vuol dire che il compilatore ed il runtime verificano la coerenza dei tipi delle variabili di un programma sia in fase di compilazione che in fase di esecuzione.
Ma cosa è un tipo? E perché è così importante?
Un tipo è semplicemente un sistema per organizzare il formato delle informazioni di un programma.
Vediamo nella tabella seguente quali sono i tipi del .NET framework:

Tipo	Valore
bool	true o false
byte	Intero positivo tra 0 e 255
sbyte	Intero tra -128 e 127
char	Un qualsiasi carattere Unicode

Tipo	Valore
DateTime	E' uno struct che rappresenta data e ora dalle 12:00:00 AM, 1/1/0001 alle 11:59:59 PM, 12/31/9999
decimal	Valore positivo e negativo con 28 digit
double	numero in virgola mobile a 64 bit
float	numero in virgola mobile a 32 bit
int	intero con segno a 32 bit (da -2.147.483.648 a 2.147.483.647)
uint	intero senza segno a 32 bit
long	intero con segno a 64 bit
ulong	intero senza segno a 64 bit
object	il tipo base da cui derivano tutti gli altri
short	intero con segno a 16 bit
string	una stringa di caratteri Unicode
TimeSpan	Un periodo di tempo positivo o negativo

Tabella 1: i tipi di dato

Nella tabella 1 sono elencati tutti i tipi di base inclusi nel .NET Framework ed il loro valore possibile. Dalla tabella si evince, ad esempio, che se dobbiamo gestire un numero superiore a 2.147.483.647, non potremmo definirlo come *int* bensì come *long*. La corretta definizione del tipo, in base alla tipologia del dato che dobbiamo trattare, è fondamentale per la corretta gestione delle variabili del programma ma anche per una migliore ottimizzazione della memoria.
La definizione di un tipo long ad esempio, riserverà in memoria lo spazio necessario a memorizzare un intero con segno a 64 bit. Spazio evidentemente sprecato per gestire, ad esempio, l'ammontare di un ordine di un sito web di commercio elettronico.
Un'errata scelta dei tipi può avere un impatto anche fortemente ne-

gativo sulle nostre applicazioni. E' quindi indispensabile analizzare correttamente le esigenze della nostra applicazione ed individuare il tipo corretto per ogni informazione che essa dovrà gestire.
Tale scelta però non deve considerarsi limitante in quanto il .NET Framework permette di effettuare delle operazioni di conversione.
Se definiamo quindi una variabile come int ed in un punto qualsiasi del nostro codice ci dovesse servire un decimal, non dobbiamo preoccuparci. Importante è, e lo ripeto, scegliere il tipo di dato corretto.
Nel prossimo paragrafo vedremo come sia possibile effettuare conversione tra i vari tipi.

2.2.1 Conversioni

Come visto nel precedente paragrafo, il .NET Framework fornisce una serie di tipi di base a cui associare le nostre variabili.
Sebbene la maggior parte dei tipi siano "concreti" e la loro descrizione rende perfettamente l'idea del tipo di dato che essi dovranno gestire, ne esiste uno più astratto denominato Object.
Questo tipo riveste particolare importanza nel .NET Framework in quanto tutti gli altri tipi "derivano" da esso (vedremo in seguito cosa vuol dire derivare). Questo vuol dire che tutti i tipi sono, alla base, un tipo Object e questo ci consente di passare da un tipo all'altro con un'operazione di conversione. Riprendendo l'esempio visto nel precedente paragrafo in cui abbiamo usato int e long, il fatto che entrambi facciano parte della famiglia Object ci consente, qualora ne avessimo la necessità, di passare senza particolari problemi dal primo al secondo tipo con un'operazione denominata conversione.Esistono però due tipi di conversione: implicite ed esplicite.Una conversione implicita è possibile quando il tipo di destinazione è sufficientemente grande per contenere il tipo di partenza.
Ad esempio,

```
byte myByte = 5;
```

```
int myInt = myByte;
```

è una conversione implicita in quanto, come evidenziato in Tabella 1, il tipo int è più grande del tipo byte e quindi può contenerlo.
Diverso è invece:

```
double myDouble = 53.751;
short myShort = myDouble; //Errore
```

Sempre facendo riferimento alla Tabella 1, il tipo double è più grande del tipo short quindi la conversione implicita non è possibile. In questo caso sarà necessaria una conversione esplicita:

```
double myDouble = 53.751;
short myShort = (short)myDouble;  //Corretto
```

Dichiarando il tipo di destinazione tra parentesi tonde (short) prima del tipo di partenza, abbiamo eseguito un'operazione denominata Casting, possibile in quanto tutti i tipi derivano da una base comune che è Object.

2.2.2 Tipi Valore e Tipi Riferimento

Prima di utilizzare un qualsiasi tipo all'interno dei nostri programmi, dobbiamo comprendere bene il modo in cui essi vengono gestiti dal .NET Framework. Esiste infatti un'importante differenza nel modo in cui, il passaggio dei tipi, viene gestito dal Common Language Runtime: i tipi infatti possono essere valore o riferimento.
Comprendere la differenza tra un tipo valore ed un tipo riferimento è un passo importantissimo per la comprensione del linguaggio e delle sue funzionalità. I tipi presentati sino ad ora sono tipi valore. La caratteristica principale di un tipo valore è che il tipo stesso contiene il valore da noi assegnato. Per comprendere meglio questa frase, è necessario comprendere come un tipo viene gestito in memoria.

Supponiamo di avere la seguente riga di codice:

```
int myVar = 42;
```

Quando l'esecuzione del nostro codice arriverà a questa istruzione, riserverà uno spazio in memoria in cui memorizzare il valore 42. Questa area di memoria prende il nome di stack.
Al contrario dei tipi valore, i tipi riferimento al loro interno contengono l'indirizzo che indica la reale posizione del dato e sono allocati in un'area di memoria denominata heap.
Sono tipi riferimento classi, delegati, array ed interfacce.
Conoscere questa differenza è molto importante al fine di prevenire errori che possono sembrare strani ed insensati.
Vediamolo qualche esempio concreto per meglio comprendere questa importante differenza.

```
namespace RefValueType {
class Program {
static void Main(string[] args) {
int a = 42;
int b = a;
Console.WriteLine("*** Esempio Value Type ***");
Console.WriteLine("Controllo variabili inserite: \r\n - a =
{0}\r\n - b = {1}", a.ToString(), b.ToString());
Console.WriteLine("Assegnazione b = 0");
b = 0;
Console.WriteLine("Controllo variabili inserite: \r\n - a =
{0}\r\n - b = {1}", a.ToString(), b.ToString());
Console.WriteLine("\r\n*** Esempio Reference Type ***");
MyReferenceTypeA myRefA = new MyReferenceTypeA();
myRefA.a = 42;
MyReferenceTypeA myRefB = myRefA;
Console.WriteLine("Controllo variabili inserite: \r\n - myRefA = {0}\r\n
```

```
  myRefB = {1}", myRefA.a.ToString(), myRefA.a.ToString());
  Console.WriteLine("Assegnazione myRefB = 0");
  myRefB.a = 0;
  Console.WriteLine("Controllo variabili inserite: \r\n - myRefA =
  {0}\r\n - myRefB = {1}", myRefA.a.ToString(),
  myRefA.a.ToString());
    }
   }
 }
```

Nella prima istruzione, assegniamo alla variabile a il valore 42. Copiamo poi la variabile a nella variabile b e stampiamo i valori a video con l'istruzione Console.WriteLine(). Come ci aspettiamo, tutti e due saranno uguali a 42.
Impostiamo poi il valore della variabile b a 0 e ristampiamo i valori. Anche in questo caso, come è facile immaginare, il valore della variabile a non cambia. Guardiamo l'output di questo semplice programma ad ulteriore conferma:

```
*** Esempio Value Type ***
 Controllo variabili inserite:
 - a = 42
 - b = 42
 Assegnazione b = 0
 Controllo variabili inserite:
 - a = 42
 - b = 0
 Premere un tasto per continuare . . .
```

Int, come abbiamo visto, è un tipo valore quindi, modificandone il contenuto, il valore di a non cambia.
Il comportamento è decisamente diverso con i tipi riferimento.
Definiamo una semplice classe in questo modo:

```
class MyReferenceTypeA{
public int a;
}
```

Le classi sono un tipo riferimento. Eseguiamo quindi le stesse operazioni di prima usando però la classe MyReferenceTypeA e non direttamente il tipo Int dell'esempio precedente:

```
MyReferenceTypeA myRefA = new MyReferenceTypeA();
myRefA.a = 42;
MyReferenceTypeA myRefB = myRefA;
Console.WriteLine("Controllo variabili inserite: \r\n - myRefA =
{0}\r\n - myRefB = {1}",
myRefA.a.ToString(), myRefA.a.ToString());
Console.WriteLine("Assegnazione myRefB = 0");
myRefB.a = 0;
Console.WriteLine("Controllo variabili inserite: \r\n - myRefA =
{0}\r\n - myRefB = {1}",
myRefA.a.ToString(), myRefA.a.ToString());
```

Nella prima istruzione creiamo una istanza di MyReferenceType chiamandola myRefA ed assegniamo al campo a il valore 42.
Creiamo una nuova istanza di MyReferenceType a cui assegneremo direttamente myRefA. A differenza di quanto visto in precedenza, se modifichiamo il valore di myRefB, essendo un tipo riferimento, verrà modificato anche il valore di myRefA.
Stampando sulla console i valori otterremo quanto segue:

```
*** Esempio Reference Type ***
 Controllo variabili inserite:
- myRefA = 42
- myRefB = 42
Assegnazione myRefB = 0
```

```
Controllo variabili inserite:
- myRefA = 0
- myRefB = 0
Premere un tasto per continuare . . .
```

Questo comportamento è dato dalla diversa natura dei due tipi. Nell'esempio appena visto, quando usiamo l'istruzione:

```
MyReferenceTypeA myRefB = myRefA;
```

stiamo semplicemente indicato a myRefB dove è memorizzato il valore di myRefA.
E' quindi naturale, in questo caso, che una modifica a myRefB vada a modificare il valore di myRefA.
Con i tipi valore questo non accadeva in quanto l'operazione:

```
int b = a;
```

effettuava a tutti gli effetti una copia del valore da un'area di memoria ad un'altra, associata alla variabile b. I tipi riferimento, a differenza dei tipi valore, sono memorizzati in un'area di memoria diversa denominata heap. Una rappresentazione molto semplificata di quello che accade con il codice dell'esempio è visibile in (Figura 2).

2.3 VARIABILI E COSTANTI

Una variabile è un elemento del nostro programma che, come dice il termine, può cambiare valore durante l'esecuzione, ad esempio per contenere il risultato di un'operazione.
Per la dichiarazione di una variabile bisogna seguire la seguente sintassi:

```
Tipo Identificatore [Inizializzatore];
```

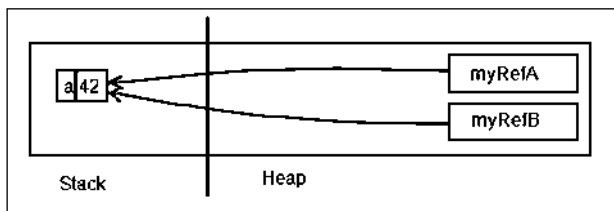



Figura 2: La differente gestione della memoria per Value Type e Reference Type

L'inizializzatore è opzionale e serve ad impostare il valore della variabile.
Ad esempio:

```
int a; //dichiariamo la variabile senza inizializzatore
int b = 42;
//dichiariamo la variabile inizializzandone il valore a 42
```

Al contrario delle variabili, le costanti sono elementi il cui valore non deve cambiare e a cui il nostro programma può fare riferimento.
Si dichiarano con la parola chiave const.
Ad esempio:

```
const int myConst = 42;
```

Sebbene la dichiarazione sia molto simile a quella di una variabile, il compilatore riconosce che il valore è costante e ne impedisce eventuali ed accidentali modifiche già in fase di compilazione.Se ad esempio facessimo:

```
myConst = 0;
```

otterremmo il seguente errore:

```
The left-hand side of an assignment must be a variable,
property or indexer
```

che ci indica il tentativo di modificare una costante.

2.4 OPERATORI ED ESPRESSIONI

Gli operatori sono elementi interni al linguaggio utili a comporre delle espressioni, il cui scopo è quello di eseguire un calcolo prestabilito. Gli operatori si dividono in 4 famiglie (che tratteremo molto velocemente):

- unari: hanno influenza su un'unica espressione e sono +, -, ++(incremento), -- (decremento), ! (negazione logica), ~ (completamento a bit).

Esempio:

```
int a = 1;
int Res = 0;
res = +a;
//risultato = 1
```

- binari: richiedono due operandi per produrre un risultato e sono: (fisibili in tabella 2)
- ternari: sono operatori che contengono tre espressioni di cui la prima deve essere booleana.

Vediamone un esempio:

```
int a = 123456789;
int b = 223456789;
string res = a == b ? "Sono uguali" : "Sono Diversi";
```

Operatore	Descrizione	Operatore	Descrizione
+	addizione	-	sottrazione
*	moltiplicazione	/	divisione
%	resto	<<	scorrimento a sinistra
>>	scorrimento a destra	==	uguaglianza
!=	disuguaglianza	<	minore di
>	maggiore di	<=	minore o uguale
>=	maggiore o uguale		
&	And	|	OR inclusivo a bit
^	OR esclusivo a bit	&&	AND Condizionale
||	OR Condizionale		

Tabella 2: Operatori binari.

```
//restituirà "sono diversi"
```

L'operatore ternario, come evidente, è molto utile per compattare il codice evitando la scrittura di un ciclo if.

- altri: rientrano in questa categoria is, as, sizeof(), typeof(), checked(), unchecked(). Operatori che non sono classificabili nei tre gruppi precedenti.

2.5 ENUM

Le enumerazioni (enum) sono un insieme di valori costanti espressi in parole anziché in numeri. Sono molto utili ai fini della comprensione del significato del valore che viene usato all'interno dei nostri programmi.
Per default, il primo elemento di un enum ha valore 0, anche se è possibile modificarlo in fase di definizione dell'enumerazione.
Il tipo di elementi che compongono un'enumerazione può essere: int, long, byte o short che va specificato anche esso in fase di

definizione dell'enumerazione.
Vediamo un semplice esempio di enumerazione che indica i giorni della settimana:

```
   namespace Enumerazioni
{
   class Program
{
   private enum Giorni
  {
   Lunedì,
   Martedì,
   Mercoledì,
   Giovedì,
   Venerdì,
   Sabato,
   Domenica
     }
  static void Main(string[] args)
  {
   Console.WriteLine
   ("Il giorno selezionato è: {0}",
   Giorni.Lunedì);
    }
   }
 }
```

La grande comodità delle enumerazioni risiede proprio nel fatto che i valori in essa contenuti possono essere richiamati usandone direttamente il nome:

```
  static void Main(string[] args)
 {
```

```
Console.WriteLine
("Il giorno selezionato è: {0}",
Giorni.Lunedì);
}
```

Microsoft Visual Studio 2005, grazie ad una comoda funzionalità denominata intellisense, ci aiuta molto anche nella scrittura del codice in presenza delle enumerazioni.
L'intellisense infatti, ricerca automaticamente l'elemento che stiamo inserendo nel codice durante la digitazione.
Di seguito un'immagine che mostra come diventi semplice scegliere i valori di una enumerazione durante la scrittura del nostro codice:

Figura 3: L'intellisense di Visual Studio quando usiamo una enum.

2.6 ISTRUZIONI DI CONTROLLO DI FLUSSO

Durante la scrittura di un software, capita molto frequentemente di dover inserire delle istruzioni che modificano il flusso di esecuzione del programma sulla base di alcune condizioni.Tali istruzioni, in generale denominate "istruzioni di controllo di flusso", sono molto usate in qualsiasi applicazione in quanto consentono di "decidere" quale istruzione eseguire in base a quello che accade nel nostro

software. Un esempio in cui tutti siamo incappati almeno una volta è l'accesso ad un'area riservata di un sito web. Inseriamo le nostre credenziali e, se sono corrette, possiamo accedere alle nostre pagine, effettuare un ordine o inserire un messaggio in un forum. Se le credenziali inserite sono errate, non potremo accedere e dovremo ripetere la procedura. In questo caso, la condizione è l'inserimento delle credenziali sulla base delle quali, verranno eseguite delle operazioni, basate appunto sulla correttezza dei dati inseriti.
Ci sono diversi tipi di istruzioni di controllo di flusso. Nei prossimi paragrafi le analizzeremo e vedremo il contesto in cui ognuna di esse viene utilizzata.

2.6.1 Istruzioni If

L'istruzione If consente di valutare un'espressione e, asseconda dell'esito, scegliere l'istruzione da eseguire.
Abbiamo a disposizione tre tipi di istruzioni If: if, if-else, if-else-if-else. L'elemento principale di un'istruzione if è l'operazione racchiusa tra le parentesi tonde.
Essa restituisce sempre un tipo booleano che sarà poi utilizzato dal costrutto per eseguire la corretta operazione
La prima delle istruzioni, la più semplice, è la seguente:

```
int a = 10;
int b = 20;
if ( a == b ){
    //fa qualcosa
}
```

L'espressione valutata dall'istruzione if è a == b che restituisce true in caso di uguaglianza e false in caso contrario. Se sono uguali, sarà eseguita l'istruzione racchiusa tra le parentesi graffe { ed }. In caso contrario, non sarà eseguito nulla in quanto l'esecuzione del codice uscirà dal ciclo if. Qualora avessimo la necessità di eseguire una par-

ticolare istruzione in caso l'espressione esaminata torni un false, allora dovremmo modificare il codice in questo modo:

```
int a = 10;
int b = 20;
if ( a == b ){
    //fa qualcosa}
else
{
    //fa qualcos'altro
}
```

In questo secondo caso, se a è diverso da b, verrebbe eseguito il codice inserito dopo l'istruzione else.
Non è raro che si abbia la necessità di eseguire ulteriori controlli in caso il primo controllo restituisca un false. Vediamo come fare:

```
int a = 10;
int b = 20;
if ( a == b ){
    //fa qualcosa
} else if (a < = b )
{
    //fa qualcos'altro
} else
{
    //fa qualcosa di diverso
}
```

e così via.
Non c'è un limite alle istruzioni else if. E comunque buona norma non inserirne troppe, pena la poca leggibilità del codice e, di conseguenza, la sua manutenzione.

2.6.2 Istruzioni Switch

Quando ci sono molte espressioni da valutare, l'istruzione if potrebbe diventare troppo lunga e poco gestibile.
Si pensi ad esempio al caso in cui volessimo dare un messaggio diverso in base al mese dell'anno in cui il programma viene eseguito: dovremmo scrivere ben 12 istruzioni if-else!
In questi casi, l'istruzione Switch ci viene in aiuto.
Essa si compone da una parola chiave switch che definisce il valore da valutare, ed una serie di istruzioni case con cui compararla.
Vediamone subito un esempio:

```
switch (Mese)
{ case "Gennaio":
Console.WriteLine("Il mese selezionato è Gennaio");
break;
 case "Febbraio":
Console.WriteLine("Il mese selezionato è Febbraio");
break;
case "Marzo":
Console.WriteLine("Il mese selezionato è Marzo");
break;
case "Aprile":
Console.WriteLine("Il mese selezionato è Aprile");
 break;
 case "Maggio":
 Console.WriteLine("Il mese selezionato è Maggio");
break; default:
 Console.WriteLine
("Il mese selezionato non è stato riconosciuto");
 break;
}
```

In questo esempio, la variabile mese sarà valutata con ogni valore di-

chiarato nell'istruzione case e, in caso di corrispondenza, sarà eseguito il codice inserito subito dopo l'istruzione di comparazione.
E' inoltre possibile definire una condizione di default da eseguire qualora nessuna delle operazioni di confronto abbia esito positivo. L'istruzione è appunto default priva di termini di paragone, ed è opzionale.

2.7 CICLI

Insieme alle istruzioni di controllo di flusso, altri tipi di istruzioni rivestono grande importanza nelle applicazioni: i cicli.
Spesso, in un programma, c'è la necessità di eseguire una stessa sequenza logica più volte. Un tipico esempio è la valutazione di una serie di elementi di una lista. Questi tipi di operazioni vengono definite cicliche e CSharp 2005 ci offre quattro tipi diversi di istruzioni, utili in altrettanti casi particolari. Si tratta delle istruzioni while, do, for e foreach.Nei prossimi paragrafi analizzeremo questi quattro tipi diversi di cicli, prestando attenzione tanto alla sintassi quanto ai casi particolari a cui essi si applicano.

2.7.1 While

Si usa il costrutto while, quando un gruppo di istruzioni deve essere eseguito in maniera continuativa fino al verificarsi di una condizione booleana. Volendo "tradurre" il ciclo while in linguaggio comune, esso assomiglierebbe ad una frase del tipo: "fai qualcosa finché questa condizione è vera".
Vediamo immediatamente un esempio di codice:

```
int myCondition = 0;
while (myCondition <= 10) {
Console.WriteLine
("Il ciclo è stato eseguito {0} volte",
myCondition.ToString());
```

```
myCondition++;
}
```

In questo esempio, l'istruzione Console.WriteLine(…) interna al ciclo, sarà eseguita finché myCondition non sarà uguale a 10.
Al verificarsi di quella condizione infatti, il flusso di esecuzione uscirà dal ciclo.I cicli while, per loro natura, nascondono un'insidia data appunto dalla condizione che deve verificarsi. Qualora essa non si verifichi mai infatti, l'esecuzione del codice non uscirà mai dal ciclo entrando in una condizione bloccante per l'applicazione denominata loop!
E' quindi molto importante assicurarsi che la condizione while si possa verificare.
Vediamo ad esempio il seguente esempio:

```
int myCondition = 0;
while (myCondition >= -1) {
Console.WriteLine
("Il ciclo è stato eseguito {0} volte", myCondition.ToString());
myCondition++;
}
```

In questo caso, myCondition sarà sempre maggiore di -1 ed il programma entrerà in loop.

2.7.2 do

Il ciclo while, come visto nel precedente paragrafo, basa l'esecuzione delle istruzioni interne su una condizione booleana. Ma cosa succede se l'esecuzione del nostro codice arriva al ciclo while con un valore che ritorna già true?
Le istruzioni interne al ciclo non saranno mai eseguite, e questa potrebbe essere anche una condizione voluta.
Ma se abbiamo la necessità di eseguire almeno una volta le istruzioni interne al ciclo, possiamo usare l'istruzione do.Il ciclo do è molto si-

mile al ciclo while con la differenza che, l'istruzione all'interno del ciclo viene eseguita almeno una volta:

```
string scelta;
 do
{
Console.WriteLine("Digita I per inserire");
Console.WriteLine("Digita C per cancellare");
Console.WriteLine("Digita U per uscire");
scelta = Console.ReadLine();
switch (scelta)
{
case "I":
//Fa qualcosa
 break;
 case "C":
 //Fa qualcosa
 break;
 case "U":
 Console.WriteLine("Ciao");
 break;
 default:
 Console.WriteLine
 ("Carattere non ammesso.
 Digita I, C o U per uscire");
 break;
   }
  } while
 (scelta != "U");
  }
```

Come evidente dal codice, le istruzioni verranno rieseguite finché non sarà scelta la lettera U.

2.7.3 for

Il ciclo for si usa quando è possibile conoscere il numero di volte in cui il ciclo deve essere eseguito. Vediamone innanzitutto la sintassi:

```
for (inizializzatore; espressione booleana; modificatore){
    //istruzione da eseguire}
```

L'istruzione interna al ciclo sarà eseguita finché l'espressione booleana è true. Vediamone un esempio pratico:

```
for (int i = 0; i < 10; i++)
{Console.WriteLine
("Il ciclo è stato eseguito {0} volte",
i.ToString()) ;
}
```

L'inizializzatore (int i = 0) viene eseguito solo al primo ciclo. L'espressione booleana (i<10), al primo ciclo sarà true (vera) quindi viene eseguita l'istruzione all'interno del ciclo.Il passo successivo sarà quello di eseguire il modificatore (i++) e rivalutare l'espressione booleana. Il ciclo si ripeterà finché essa tornerà false.

2.7.4 foreach

Il ciclo foreach è molto utile quando bisogna iterare gli elementi di una collection (che vedremo nel capitolo 9) .La sintassi di un ciclo foreach è la seguente:

```
foreach (tipo nella collection){
    //istruzione da eseguire
}
```

Torneremo su questo ciclo nel capitolo dedicato alle collection.

CLASSI ED OGGETTI

Non si può parlare di C# o di .NET Framework senza parlare di programmazione ad oggetti. Questa metodologia di sviluppo ha praticamente rivoluzionato tanto il modo di sviluppare le applicazioni, tanto il modo in cui "pensarle" e progettarle.
In questo capitolo vedremo i mattoni fondamentali della programmazione ad oggetti: le classi ed, ovviamente, gli oggetti. Ci concentreremo sulla loro struttura, sul loro scopo, sulla loro creazione e sul loro utilizzo. Quando ci si avvicina alla programmazione ad oggetti, comprendere questi concetti può sembrare ostico e complesso. In realtà, i concetti che andremo a spiegare sono solo diversi da quella che è stata per anni l'unica forma di programmazione, ovvero quella procedurale. Comprendere la programmazione ad oggetti ed usarla, porta con se numerosi vantaggi che saranno compresi man mano che questa metodologia sarà adoperata nell'uso quotidiano.
Una volta compreso l'uso di tali elementi, ci dedicheremo, nel prossimo capitolo, a quelli che vengono definiti i paradigmi della programmazione ad oggetti.

3.1 OGGETTI E CLASSI

Nel capitolo 2, quando abbiamo dato un primo sguardo ad un programma scritto in C#, abbiamo incontrato la parola chiave class. Sinteticamente si era detto che una classe è un elemento usato da C# per creare un oggetto. Ma cosa è un oggetto?
Per comprenderlo, non c'è niente di più facile che fare un parallelismo con il mondo "reale". In questo momento avete in mano un libro. Un libro è un oggetto dotato di alcune caratteristiche come numero di pagine, numero di capitoli etc. !
Se ci guardiamo intorno, siamo circondati da moltissimi oggetti, di varia forma e complessità. Ogni oggetto che ci circonda ha delle caratteristiche che lo contraddistingue dagli altri. Una sedia, ad esempio, ha degli attributi e dei comportamenti certamente diversi da

quelli di un’automobile.Prendiamo come esempio appunto un’automobile.Essa avrà certamente un attributo che ne specifica in numero di ruote e sarà di tipi int. Un altro attributo, sempre int, potrebbe essere il numero di marce.
Pensando ancora agli attributi, potremmo averne uno ti tipo string per il colore ed ancora un bool per la presenza degli air bag e così via.
Pensiamo ora al comportamento. Un’auto andrà certamente in avanti ed indietro, sterzerà, frenerà etc.
Tutte le caratteristiche che abbiamo elencato, sono sostanzialmente attributi e comportamenti che sono applicabili al tipo di oggetto automobile. Anche la nostra sedia sarà dotata di attributi e comportamenti, sicuramente diversi da quelli dell’auto, ma che ne descriveranno il modo in cui è fatta e l’utilizzo che se ne deve fare.
In CSharp, come per gli oggetti “reali” come l’automobile, abbiamo la necessità di descrivere un oggetto assegnandogli degli attributi e dei comportamenti. Solo quando l’oggetto sarà ben definito sarà possibile crearlo ed usarlo.
Per definire un oggetto in CSharp si usa la parola chiave class. Una classe è praticamente il “progetto” di un oggetto che sarà poi creato durante l’esecuzione del nostro software.
La differenza che esiste quindi tra una classe ed un oggetto è che l’oggetto esiste solo a runtime (ovvero durante l’esecuzione del programma) e sarà creato sulla base della descrizione fornita dalla classe.Per meglio chiarire questi aspetti, vediamone subito degli esempi pratici.

3.1.1 Definizione di una classe

Riprendendo l’esempio del paragrafo precedente relativo all’automobile, cerchiamo di tradurre quello che abbiamo visto in codice.

```
class Automobile {
/// <summary>
/// Campi della classe Automobile
```

```
/// </summary>
public int _nRuote;
public int _nMarce;
public int _nPorte;
public int _VelocitàMax;
public string _Marca;
public string _Modello;
public string _Colore;
public bool _AirBag;
public bool _ABS;
public bool _Clima;
/// <summary>
/// Costruttore dell'oggetto automobile
/// </summary>
public Automobile(int NumeroRuote, int NumeroMarce,
int NumeroPorte, int VelocitàMax, string Marca, string Modello,
string Colore, bool AirBag, bool ABS, bool Clima) {
_nRuote = NumeroRuote;
_nMarce = NumeroMarce
_nPorte = NumeroPorte;
_VelocitàMax = VelocitàMax;
_Marca = Marca;
_Modello = Modello;
_Colore = Colore;
_AirBag = AirBag;
_ABS = ABS;
_Clima = Clima;
}
//Aumenta la marcia attuale passata come argomento al metodo
private int AumentaMarcia(int MarciaAttuale) {
if( MarciaAttuale == _nMarce ) {
return MarciaAttuale;
} else {
```

```
return MarciaAttuale++;
  }
    }
//Riduce la marcia attuale passata come argomento al metodo
private int RiduciMarcia( int MarciaAttuale ) {
if( MarciaAttuale == 0) {
return MarciaAttuale;
 } else {
return MarciaAttuale--;
 }
  }

private void Accellera() {
//codice per far accellerare l'auto
 }

private void Frena() {
//codice per far freanre l'auto
}
}
```

La prima operazione da fare è quella di definire la nostra classe utilizzando la parola chiave class seguita dal nome che vogliamo dargli. Nel nostro caso: class Automobile (riga 1).
Se il nostro oggetto ha dei campi, per questioni di leggibilità li inseriamo subito dopo la dichiarazione della classe.
Un aspetto molto importante da valutare è la visibilità che ogni componente della nostra classe (in questo caso i campi), deve avere rispetto al resto del nostro programma.
In questo esempio, per semplicità, alla dichiarazione del tipo di campo è stata fatta precedere la parola chiave public. Essa fa sì che il nostro campo sia visto (e modificato) anche dal codice esterno alla classe stessa. Sebbene questa situazione può essere comoda, non è

sempre la più indicata.
Gli ambiti di visibilità (tecnicamente chiamati modificatori) che possono essere assegnati agli elementi di una classe (compresa la classe stessa) sono:

- public: visibile da qualsiasi elemento, anche esterno alla classe.
- private: visibile sono all'interno della stessa classe.
- prtected: visibile sono all'interno della stessa classe o delle classi derivate (vedi Capitolo 5 su Ereditarietà e Poliformismo).
- internal: visibile dai membri dello stesso programma.
- protected internal: visibile dai membri dello stesso programma o di membri delle classi derivate.

Scegliere il modificatore corretto è abbastanza semplice: la cosa importante è avere chiaro innanzitutto lo scopo della nostra classe e successivamente quello dell'elemento (campo, metodo etc.) che stiamo definendo.
Nel codice sopra riportato, dalla riga cinque alla quattordici vengono definiti tutti i campi che il nostro oggetto Automobile avrà.
Alla riga diciotto è stato introdotto un elemento importante: il costruttore del nostro oggetto. Un costruttore è, come dice il termine, un elemento utile in fase di esecuzione del programma alla creazione dell'oggetto reale a partire dagli elementi definiti nella nostra classe.
Un costruttore si definisce come un metodo particolare che ha lo stesso nome della classe e non ha tipo di ritorno (vedremo meglio questo aspetto nei prossimi paragrafi).
Questo particolare metodo sarà chiamato solo nel momento in cui il nostro programma avrà la necessità di costruire l'oggetto.
Nel nostro caso specifico, il costruttore accetta in ingresso i valori relativi a tutti i campi della classe Automobile e, al suo interno, non fa altro che assegnare i valori passati ai relativi campi.
Il nostro oggetto è così completo ma, tornado al paragone dell'au-

tomobile fatto in precedenza, così com'è non può fare nulla.
Non è stato definito alcun comportamento!
I "comportamenti" di un oggetto sono definiti metodi, e sono quelli che definiamo dalla riga 53.
Prendiamo ad esempio il primo metodo denominato AumentaMarcia. Come è facile immaginare, esso rappresenta il comportamento di un'automobile quando appunto aumentiamo la marcia. Nel caso specifico dell'esempio, un controllo if valuta se la marcia che cerchiamo di mettere è uguale al numero di marce che la nostra automobile ha. Se così fosse, avremmo raggiunto la marcia più alta a disposizione. In caso contrario, il nostro metodo ci restituirà la marcia successiva. Lo stesso ragionamento è stato fatto per gli altri metodi.
La classe riportata in alto è solo un esempio.
Difficilmente vi sarà chiesto di implementare un oggetto automobile come sopra. Dato però il forte parallelismo con un oggetto "reale", ritengo che sia un buon modo per avvicinarsi alla programmazione ad oggetti.

3.1.2 Creazione un oggetto

Nel precedente paragrafo abbiamo visto la definizione di una classe e capito a cosa serve e qual è la differenza tra una classe ed un oggetto. In questo paragrafo, vedremo come creare ed usare un oggetto, usando la classe di prima come base.
Per l'esempio è stata utilizzata un'applicazione di tipo console che, a fronte delle risposte date ad alcune domande, costruirà il nostro oggetto Automobile. Facendo riferimento alla classe del paragrafo precedente, alla riga 18 troviamo il costruttore dell'oggetto. Tale costruttore vuole in ingresso una serie di parametri che prepareremo ponendo delle domande all'utente. Il codice completo della applicazione di test è consultabile nel file allegato. Diamo comunque uno sguardo ad una sua sintesi:

```
private static void CreazioneAuto() {
```

```
Console.WriteLine("*** Esempio di oggetto auto ***");
Console.Write( "Immetti il numero di ruote: " );
int _nRuote = int.Parse( Console.ReadLine() );
```

Altre istruzioni per impostare i valori dei parametri

```
Console.WriteLine( "*** Creazione dell'automobile ***" );
Automobile auto = new Automobile
( nRuote, _nMarce, _nPorte, _VelocitàMax, _Marca, _Modello,
_Colore, _AirBag, _ABS, _Clima );
Console.Write
( "Automobile {0} creata. Vuoi vederne i dettagli? (S/N)", auto._Marca );
}
```

Abbiamo tutti i valori utili al costruttore per creare l'oggetto e valorizzarne i campi, non ci resta quindi che richiamarlo con la sintassi

```
Tipo NomeOggetto = new Costruttore();
```

Nel nostro esempio, il costruttore vuole dei parametri in ingresso. Parametri che abbiamo preparato ponendo le domande del codice precedente all'utente:

```
Automobile auto = new Automobile( _nRuote, _nMarce,
_nPorte, _VelocitàMax, _Marca, _Modello, _Colore, _AirBag,
_ABS, _Clima );
```

Cosa succede sinteticamente all'interno dell'ambiente di esecuzione?Come abbiamo visto nel paragrafo 3.1.2, gli oggetti sono di tipo riferimento. Il CLR ha quindi creato un'area in una zona di memoria chiamata heap da dedicare all'oggetto auto.
Il costruttore valorizza i campi della classe memorizzandone, in un certo senso, i valori nell'area di memoria che il CLR le ha riservato crean-

do, a tutti gli effetti, un'istanza della classe Automobile. L'oggetto è stato quindi creato realmente sfruttando le indicazioni fornite dalla classe.Da questo momento in poi, auto (che è l'istanza della nostra classe Automobile) contiene un riferimento a quella specifica area di memoria creata dal CLR.
Ogni volta che, nel nostro codice, ci riferiremo ad auto, faremo riferimento ad un indirizzo specifico dell'heap.
Questo dovrebbe chiarire anche il comportamento particolare dei tipi riferimento spiegati nel paragrafo 3.1.2.
Se noi creassimo quindi un altro oggetto richiamando nuovamente il costruttore della classe Automobile, i passi elencati si ripeterebbero, creando alla fine una nuova area di memoria ed un nuovo riferimento ad essa. Un altro oggetto insomma.
A questo punto dovrebbe essere chiara la differenza tra classi ed oggetti. Uno degli esempi più esplicativi per comprendere la differenza tra i due elementi cita: la classe è il progetto di un oggetto.
Con un progetto di un'auto, se ne possono creare molte.

3.1.3 Uso dei metodi

Nei precedenti paragrafi abbiamo visto come creare una classe ed istanziarla per creare i nostri oggetti. Ma nel paragrafo 4.1.1 avevamo implementato anche dei comportamenti (metodi).
Un metodo è semplicemente un comportamento che può avere un oggetto.
Un metodo è composto dalla seguente sintassi:

```
Modificatore TipoDiRitorno Nome (Parametri){
    //implementazione
}
```

Il modificatore assegnabile ad un metodo è dello stesso tipo già visto nel paragrafo 4.1.1.
Il tipo di ritorno identifica appunto il tipo di dato che dovrà essere re-

stituito al chiamante quando l'esecuzione del metodo sarà completata. Il tipo di ritorno può essere qualsiasi cosa come i tipi nativi visti nel capitolo 3.1 alla tabella 1, un qualsiasi oggetto del .NET Framework od un nostro tipo personalizzato (come ad esempio lo stesso oggetto Automobile).
Ma un metodo può anche eseguire delle operazioni al suo interno che non hanno la necessità di ritornare alcun dato. Ad esempio, se abbiamo implementato un metodo per scrivere dei dati su un documento di testo, il compito del metodo sarà quello di scrivere l'informazione passata come parametro ma non restituire nulla al chiamante.
Per identificare questi tipi di metodi si usa il tipo di ritorno void:

```
private void ScriviLog (string Messaggio){
    //implementazione della scrittura su file di testo
}
```

Vediamo ora come usare i metodi.
Dopo aver creato l'oggetto con l'istruzione:

```
Automobile auto = new Automobile( _nRuote, _nMarce,
_nPorte, _VelocitàMax, _Marca, _Modello, _
Colore,
_AirBag, _ABS, _Clima );
```

l'accesso ai membri pubblici degli oggetti avviene con la sintassi:

```
NomeOggetto.Membro
```

Se ad esempio volessimo elencare il valore di tutti i campi che compongono l'oggetto auto, dovremmo fare:

```
Console.WriteLine( "*** Dettagli auto {0} {1}: ",
```

```
auto._Marca, auto._Modello );
Console.WriteLine( "\tNumero ruote: {0}", auto._nRuote );
Console.WriteLine( "\tNumero marce: {0}", auto._nMarce );
Console.WriteLine( "\tNumero porte: {0}", auto._nPorte );
```

E così via per tutti gli altri campi.
Allo stesso modo possiamo accedere ai metodi di cui il nostro oggetto è dotato.
Se, ad esempio, volessimo incrementare la marcia della nostra auto (metodo AumentaMarcia descritto alla riga 31), dovremmo usare la sintassi:

```
auto.AumentaMarcia( MarciaAttuale );
```

E così per tutti gli altri metodi pubblici. Gli oggetti di un software possono quindi essere paragonati a tutti gli effetti agli oggetti reali.
Si costruiscono sulla base di un progetto (la classe), si usano sfruttandone il loro comportamento (metodi) e si "distruggono" (è il termine tecnico che indica la cancellazione dell'istanza dell'oggetto dall'area di memoria assegnata dal CLR).

3.2 MEMBRI STATICI

Nel precedente paragrafo abbiamo visto come, richiamando il costruttore di una classe, ne creiamo una sua istanza in memoria.
Questo vuol dire che, se creiamo una nuova istanza, essa occuperà un'altra area di memoria essendo, a tutti gli effetti, un'entità diversa dalla precedente.
Può capitare però di avere la necessità di avere una sola copia di un membro di una classe, indipendentemente da quante siano le copie in memoria.
A questo scopo ci viene in aiuto la parola chiave static.
Un membro statico, in sostanza, non viene istanziato ed è accessibile

a prescindere dall'istanza della classe stessa.
Supponiamo ad esempio di avere la necessità di generare delle password da dare ad un utente per l'accesso ad un software o ad un sito web. Creiamo quindi un nuovo oggetto che si occupa di svolgere tale compito. Riceverà quindi uno UserName e, sulla base di esso, restituirà una password.
Come è facile intuire, il nostro oggetto GeneraPassword non ha la necessità di essere istanziato. Il suo compito è tutto sommato semplice e, nel caso dovessimo generare numerose password, non avrebbe senso creare altrettante istanze dello stesso oggetto in memoria.
Di seguito un breve esempio di come potrebbe essere strutturato questo oggetto:

```
public class GeneraPassword {
public static String Password(string Nome) {
string Data = DateTime.Now.Millisecond.ToString();
string prePassword = Nome + Data;
string password = prePassword.GetHashCode().ToString();
return password;
}
}
```

Sebbene questa classe, come struttura, assomiglia alla classe d'esempio vista in precedenza, essa si differenzia per l'assenza di un costruttore e per il metodo Password preceduto dalla parola chiave static. Per usare tale metodo basterà scrivere:

```
Console.WriteLine(GeneraPassword.Password("Mighell"));
```

I membri statici si prestano a numerosi scopi consentendo un notevole risparmio di memoria impegnata. Negli esempi fatti fino ad ora, abbiamo spesso usato questo tipo di membri. Console.WriteLine() ad esempio. WriteLine è semplicemente un metodo statico della classe

Console. Si potrebbe essere portati a pensare però, che l'utilizzo di metodi statici sia preferibile all'utilizzo dei normali metodi che hanno la necessità di un'istanza della classe. Questo è sbagliato.
Ci sono contesti (come WriteLine) in cui l'utilizzo di un membro statico è decisamente preferibile. Ma non dimentichiamoci che l'area di memoria assegnata a tali membri è condivisa. Se pensiamo ad un ambiente multiutente, come ad esempio un sito web, non è pensabile che una variabile assegnata ad un utente sia letta o assegnata, anche accidentalmente, da un altro.

3.3 DISTRUZIONE DEGLI OGGETTI

Se nel momento in cui istanziamo un oggetto, ne creiamo una rappresentazione in memoria, la distruzione di un oggetto consiste praticamente nel liberare tutte le risorse da esso occupate sul sistema. In particolare, la distruzione di un oggetto comporta il rilascio dell'area di memoria in cui esso era allocato.Chi ha lavorato in passato con altri linguaggi e sistemi sa quanto fossero problematici i problemi di gestione della memoria. Il .NET Framework, sin dalla prima versione, introduce una procedura denominata Garbage Collection attraverso il quale vengono liberate le risorse non più utilizzate dal programma.Il Collector, che è l'oggetto che si occupa autonomamente di liberare la memoria, ha un funzionamento abbastanza complesso basato su algoritmi interni che lo rendono molto performante.
Spiegare nel dettaglio questi meccanismi richiederebbe parecchio spazio e la conoscenza di alcune nozioni non trattabili in questo testo.Si rimanda quindi alla abbondante documentazione disponibile on line per gli approfondimenti.

3.4 PARAMETRI

Nell'esempio relativo alla classe Automobile abbiamo già toccato l'argomento dei parametri. Il costruttore della classe infatti (riga 18),

accettava in ingresso dei dati utili a valorizzare, in fase di costruzione dell'oggetto, tutti i suoi campi. I parametri rappresentano sostanzialmente il meccanismo con il quale viene specificato il tipo di variabili che un metodo accetta dai relativi chiamanti.
I tipi di parametri che possiamo utilizzare sono: valore, riferimento, out e params. Vediamoli insieme.

3.4.1 Parametri Valore

Come abbiamo visto nel capitolo 3.1.2, esistono dei tipi valore e dei tipi di riferimento. Per default, quando ad un metodo passiamo un tipo valore, esso sarà trattato appunto come valore.
Se passiamo un tipo riferimento sarà trattato di conseguenza.
Prendendo sempre l'esempio della classe Automobile, nel costruttore vengono passati una serie di parametri che sono, per loro natura, tipi valore.
Questo vuol dire che, a livello di memoria, il loro contenuto verrà copiato nel metodo che stiamo richiamando (nel nostro caso il costruttore della classe).
Il comportamento dei parametri passati come valore è identico a quello classico: se modifichiamo il contenuto di un tipo valore, il contenuto del tipo originale non sarà modificato.

3.4.2 Parametri Riferimento

Come per i parametri passati come valore, il comportamento dei parametri di riferimento è uguale a quello dei corrispettivi tipi. Di default, se ad un metodo passiamo un tipo riferimento (un oggetto ad esempio) esso sarà considerato come tale.
Può però essere necessario passare un tipo che normalmente è di tipo valore, come riferimento.
Per farlo, è necessario specificare, nel corpo del metodo, che il tipo che stiamo passando deve essere considerato riferimento:

```
public void myTestMethod (ref string test){
```

```
    //corpo del metodo
}
```

ed in chiamata:

```
myClass.myTestMethod(ref myString);
```

In questo modo, sebbene la stringa myString sia per definizione un tipo valore, se la modifichiamo all'interno del metodo myTestMethod, il suo valore sarà modificato anche nella variabile originale.

3.4.3 Parametri Out

Come abbiamo visto nel capitolo 4.1.3, i metodi possono essere dotati di un tipo di ritorno che rappresenta l'elemento da restituire al chiamante quando l'esecuzione è completata.
Da un metodo può tornare un solo tipo. Ci sono casi però in cui questo comportamento può essere una limitazione. Per superarla, ci vengono in aiuto i parametri out.
Un parametro di tipo out viene valorizzato all'interno del metodo e può essere utilizzato direttamente dal chiamante.
Vediamone un esempio:

```
public static void myTestMethod(out int Param1)
{Param1 = 100;
}
```

Come evidente, il myTestMethod è di tipo void, quindi non ritorna alcun valore, ma accetta un parametro di tipo out.

```
  static void Main()
{
  int myValue;
  myTestMethod(out myValue);
```

```
  Console.WriteLine(myValue.ToString());
}
```

Quando richiamiamo un metodo che accetta un parametro out, dobbiamo innanzitutto definire il parametro senza valorizzarlo e successivamente inviarlo al metodo facendolo precedere dalla parola chiave out.
Alla fine dell'esecuzione del metodo, myValue sarà valorizzata con il valore assegnato dal metodo.

3.4.4 Parametri Params

Quando ad un metodo è necessario passare numerosi valori o non se ne può conoscere in anticipo il loro numero, può essere d'aiuto il tipo Params. Questi permettono di passare come tipo un array (che vedremo nel capitolo 9) o elenchi di valori e deve essere l'ultimo nell'elenco dei parametri accettati da un metodo.
Ad esempio:

```
public void myTestMethod (string test, params string[] myStrings){
    //corpo del metodo
}
```

3.5 OVERLOADING DEI METODI

L'overloading è la capacità di un programma di definire più metodi che hanno lo stesso nome ma che differiscono per il tipo e/o il numero dei parametri che accettano in ingresso.
Tale funzionalità del .NET Framework è molto utile in fase di sviluppo di un'applicazione in quanto ci permette di mantenere ordine nei nomi degli elementi che compongono il nostro software.
Vediamo subito un esempio concreto.
Supponiamo di aver realizzato un software che ha la necessità di validare le credenziali di un utente prima di essere eseguito.

A tale scopo implementiamo un metodo denominato GetUser con lo scopo di recuperare i dati dell'utente da una fonte dati (un Data Base ad esempio).
Tale metodo accetterà in ingresso un nome utente ed una password e restituirà un oggetto utente:

```
public User GetUser(string UserName, string Password)
{//Esegue un'interrogazione al Data Base.
 //Se i dati sono corretti, ritornerà un oggetto User
 //ed il login avrà esito positivo.
}
```

Ora, supponiamo che nel nostro codice ci sia la necessità di recuperare i dati di un utente specifico (magari un cliente). Un nome descrittivo per il nostro nuovo metodo potrebbe essere ancora GetUser ma che abbiamo già implementato per scopi diversi.Ecco quindi che ci torna utile l'overloading. Possiamo creare un nuovo metodo che ha lo stesso nome del precedente ma che accetta parametri diversi. Due ipotetici metodi potrebbero essere:

```
public User GetUser(int IDUser)
{//Esegue un'interrogazione al Data Base.
//Se i dati sono corretti,
ritornerà un oggetto User
return null;}
```

e:

```
public User GetUser(string Email)  {
//Esegue un'interrogazione al Data Base.
//Se i dati sono corretti, ritornerà un oggetto User
return null;
}
```

Abbiamo ora tre diversi metodi con lo stesso nome ma con parametri diversi.
Sarà cura del runtime, in base ai parametri passati, scegliere quale dei metodi chiamare.
E' importante sottolineare una cosa importante: l'unico modo per fare l'overloading di un metodo è cambiare il tipo o il numero dei parametri che esso accetta in ingresso. Non è possibile ad esempio scambiarli solo di ordine o modificare il tipo di ritorno lasciando inalterati i parametri.

3.6 PROPRIETÀ

Le proprietà sono membri di una classe che hanno lo scopo di fornire un acceso specializzato ai campi della classe stessa. Prendiamo ad esempio una classe denominata myClass che contiene un campo denominato Description. Vogliamo che tale campo sia accessibile dall'esterno della classe ma il suo contenuto deve essere definito dalla classe stessa.
Se dichiarassimo il suo modificatore a public, come visto nel paragrafo 4.1.1, daremmo la possibilità a tutto il resto del programma di accedere e definire il nostro campo Description, perdendo di fatto la caratteristica che volevamo.
Quello che dobbiamo fare è incapsulare il capo in una proprietà.
Una proprietà si definisce con un modificatore public, il tipo che vogliamo restituire e due metodi di accesso facoltativi denominati Get e Set:

```
class Proprietà {
private string _Descrizione;
public string Descrizione {
get {
return _Descrizione;
 }
```

```
set {
_Descrizione = value;
}
 }
}
```

Questa proprietà molto semplice ha il solo scopo di rendere accessibile dall'esterno il campo _Descrizione (privato) e, apparentemente, le proprietà avrebbero poco senso.
Ma proviamo ad immaginare se _Descrizione deve essere costruito all'interno della classe:

```
class Proprietà {
private string _Descrizione;
public string Descrizione {
get {
_Descrizione = "Campo richiesto il " +
DateTime.Now.ToString();
return _Descrizione;
}
set {
Descrizione = value;
}
}
}
```

In questo esempio, il contenuto di _Descrizione viene letteralmente costruito al momento della richiesta, nascondendone l'implementazione al chiamante. In questo modo, il campo privato resterà accessibile solo all'interno della classe che lo contiene, mentre pubblicamente sarà costruito a richiesta in base alla nostra implementazione specifica.
I metodi di accesso Get e Set sono alternativamente opzionali. Questo vuol dire che potremmo creare proprietà di sola lettura ometten

do il Set e, potenzialmente, proprietà di sola scrittura omettendo il metodo Get.
Ad esempio:

```
class Proprietà {
private string _Descrizione;
public string Descrizione {
get {
return _Descrizione;
  }
 }
}
```

PRINCIPI DI OBJECT ORIENTED PROGRAMMING

Nei precedenti capitoli, abbiamo analizzato molti degli aspetti basilari del linguaggio CSharp e del suo ambiente di esecuzione.
In questo capitolo analizzeremo invece uno dei pilastri della programmazione moderna: la programmazione ad oggetti. I concetti spiegati in questo capitolo, sebbene corredati da esempi in CSharp, sono generali ed applicabili a tutti i linguaggi orientati agli oggetti.
Tratteremo i quattro paradigmi fondamentali di questa metodologia di programmazione: incapsulamento, ereditarietà, poliformismo e astrazione. Tutti concetti che, al giorno d'oggi, non si possono tralasciare. La maggior parte dei software oggi presenti sul mercato usa questa metodologia di sviluppo. Data la natura di questo libro, non sarà possibile approfondire tutti gli aspetti legati alla programmazione ad oggetti. Essendo comunque un argomento importante, utile anche alla comprensione del resto del libro, se ne tratteranno gli aspetti più importanti cercando di dare gli strumenti di base per iniziare.

4.1 PENSARE AD OGGETTI

Programmare ad oggetti non vuol dire solo conoscerne ed applicarne i principi, ma "pensare" ad un'applicazione secondo i paradigmi dell'OOP (Object Oriented Programming).
Concepire un'applicazione secondo la programmazione procedurale, renderebbe praticamente inutile, se non proprio inapplicabile, l'utilizzo degli oggetti e dei vantaggi che da essi derivano. Un'applicazione pensata e progettata secondo i paradigmi dell'OOP, sarà infatti composta da un insieme di oggetti, tutti indipendenti ma che possono comunicare tra loro.
Pensare ad oggetti vuol dire individuare, all'interno dell'applicazione che si stà andando a sviluppare, tutti gli oggetti, le eventuali famiglie a cui essi appartengono, le responsabilità individuali e, per fi-

nire, il modo in cui essi interagiscono tra loro. Iniziamo questo argomento parlando dell'incapsulamento.

4.2 INCAPSULAMENTO

L'incapsulamento (o "information hiding") è la capacità di un oggetto di nascondere la sua implementazione interna rispetto al resto del codice. Per meglio comprendere questo concetto, immaginiamo ad esempio un'applicazione per la gestione dei conti correnti bancari. Dopo aver letto il paragrafo precedente, dovrebbe essere chiaro che un conto corrente può essere immaginato come un oggetto il cui compito sarà evidentemente quello di rappresentare, per il resto del software, i conti correnti dei clienti della banca.
Chiameremo l'oggetto Conto. Al suo interno ci saranno sicuramente dei campi che identificano il proprietario del conto, il saldo, il tasso di interesse, la data di apertura etc.
L'oggetto Conto sarà anche dotato di suoi comportamenti specifici (metodi) come Apri, Chiudi, Versa, Preleva etc.
Il codice del nostro oggetto conto potrebbe essere il seguente:

```
class Conto {
private double _Saldo;
private double _TassoInteresse;
private string _Proprietario;
private DateTime _DataApertura;

public double Saldo {
get { return _Saldo; }
  }
public double TassoInteresse {
get { return _TassoInteresse; }
set { _TassoInteresse = value; }
   }
```

```
public string Proprietario {
get { return _Proprietario; }
set { _Proprietario = value; }
 }
public DateTime DataApertura {
get { return _DataApertura; }
set { _DataApertura = value; }
  }
public void Apri() {
 //Implementazione del metodo Arpi
  }
 public void Chiudi() {
 //Implementazione del metodo Chiudi
  }
 public double Versa(double importo) {
 _Saldo += importo;
return Saldo;
     }
 public double Preleva(double importo) {
 _Saldo -= importo;
 return Saldo;
  }
 }
}
```

I campi sono tutti privati ed accessibili dall'esterno grazie alle rispettive proprietà. Il campo _Saldo invece, come evidente dal codice, è accessibile in sola lettura dall'esterno in quanto la relativa proprietà non ha il setter (Set).
In questo modo, la logica di calcolo del saldo è nascosta all'esterno. Il compito di calcolare ed assegnare un valore al Saldo infatti è demandata all'oggetto Conto. Sarebbe insensato permettere tale modifica dall'esterno dell'oggetto.

Se il campo _Saldo fosse impostabile all'esterno, o per errore o per malizia, potremmo fornire un'informazione errata al cliente o ad un livello più alto dell'applicazione.
E' evidente che tale comportamento è inaccettabile, quindi impediamo a priori tale modifica dall'esterno.
Un altro vantaggio non meno trascurabile è relativo alla manutenzione del codice. Se infatti i criteri del calcolo del saldo dovessero essere modificati nel corso del tempo, si dovrà intervenire solo in questo oggetto, modificando la relativa proprietà.
Grazie all'incapsulamento quindi, la gestione, la manutenzione e, sotto certi aspetti anche la sicurezza del nostro codice sono migliorate. Per tutto il resto del software infatti, visualizzare il saldo di uno specifico conto corrente si tradurrà in un'istruzione del tipo:

```
conto.Saldo;
```

senza che vi sia la necessità di conoscerne l'implementazione.

4.3 EREDITARIETÀ

L'ereditarietà rappresenta il meccanismo in base al quale è possibile creare nuovi oggetti a partire da oggetti già esistenti.
In breve, una classe figlia, denominata classe derivata, condivide caratteristiche e funzionamento della classe padre detta classe base.
La relazione tra classe derivata e classe base si definisce, in linguaggio comune, "è un" in quanto la classe derivata è, a tutti gli effetti, una classe base di cui può modificarne caratteristiche e comportamenti.
Il vantaggio fondamentale dell'ereditarietà è dato dalla possibilità di riutilizzare parecchio codice già scritto per un'altra classe.
Il modo migliore per capire il principio dell'ereditarietà è fare riferimento, ancora una volta, al mondo reale ed in particolar modo alla natura. Prendiamo ad esempio l'uomo: è un mammifero, così come lo è anche un cane o un gatto. I mammiferi hanno caratteristiche in

comune come essere a sangue caldo etc.
Ma a loro volta, le varie tipologie di mammiferi hanno caratteristiche proprie come ad esempio parlare per l'uomo, avere quattro zampe per il cane etc.
Riepilogando, l'uomo "è un" mammifero. Mammifero può quindi rappresentare la classe base e Uomo quella derivata.
Torniamo alla programmazione ad oggetti e riportiamo il concetto appena descritto sul piano del codice.
Se stiamo realizzando un applicativo gestionale, dovremo sicuramente trattare sia clienti, fornitori ed agenti.
Il fornitore è un'entità a se stante (può essere un'azienda), ma clienti ed agenti saranno (si spera) delle persone.
E' già evidente un rapporto di appartenenza delle due entità ad un tipo comune (Persona).
Quello che potrebbe differenziare i clienti dagli agenti potrebbe essere, ad esempio, un credito per i clienti ed un'area di competenza per gli agenti.
Vediamo come rappresentare questa appartenenza in codice.
Innanzitutto definiamo una classe persona che raccoglierà tutte le caratteristiche comuni tra le due entità (la classe è molto semplice ed è da intendersi solo a titolo di esempio):

```
class Persona {
private string _Nome;
private string _Cognome;
private int _Età;
private string _Indirizzo;
private string _Telefono;
public string Nome {
 get { return _Nome; }
 set { _Nome = value; }
  }
public string Cognome {
```

```
    get { return _Cognome; }
    set { _Cognome = value; }
        }
    public Persona() {
    }
  }
}
```

Nella classe persona, e più in generale in una classe base, inseriamo tutte le caratteristiche comuni che caratterizzando una famiglia di oggetti che andremo ad utilizzare all'interno del nostro codice.
Pronta la classe base, creiamo la prima classe derivata chiamata Cliente. Il rapporto di ereditarietà dalla classe Persona si definisce con i due punti (:).

```
class Cliente : Persona {
private double _Credito;
public double Credito {
get { return _Credito; }
set { _Credito = value; }
 }
public Cliente() {
  }
}
```

Lo stesso discorso va fatto per la classe Agente:

```
class Agente : Persona {
private string _AreaDiCompetenza;
public string AreaDiCompetenza {
get { return _AreaDiCompetenza; }
set { _AreaDiCompetenza = value; }
  }
```

```
public Agente() {
  }
}
```

Come evidente, nelle due classi derivate non sono stati ridefiniti i campi comuni come Nome, Cognome etc. Le due classi derivate però, quando saranno utilizzate, si comporteranno come quei campi fossero stati definiti al loro interno:

```
class Test {
private Test() {
Cliente cliente = new Cliente();
Agente agente = new Agente();
cliente.Nome = "Michele";
agente.Nome = "Giuseppe";
 }
}
```

La prima cosa evidente è il risparmio di codice. Senza ereditarietà, avremmo dovuto ridefinire tutti i campi in comune. Altro vantaggio non indifferenze è la pulizia del codice. Guardando la classe Agente infatti, è già evidente quanto essa sia compatta e facile da gestire. Inoltre, analizzando a posteriori il nostro codice, non ci troveremmo di fronte a decine di campi denominati ad esempio Cognome senza sapere di preciso a quale classe essi appartengono.
Dall'analisi dell'ereditarietà nasce però una considerazione molto importante, già accennata nel paragrafo 5.1: bisogna pensare ad oggetti.

4.4 POLIFORMISMO

Il poliformismo è la capacità di una classe di assumere letteralmente più forme, ridefinendo il comportamento delle sue proprietà e dei

suoi metodi.Il concetto può sembrare un po' astratto quindi, per comprenderlo meglio, ci rifacciamo ancora una volta ad un paragone con il mondo reale. Nel paragrafo 5.3 relativo all'ereditarietà, abbiamo visto come un uomo ed un cane "derivano" da un raggruppamento base denominato Mammifero, in quanto le due entità hanno dei tratti in comune. Uno dei comportamenti (metodi) di un'ipotetica classe base Mammifero, potrebbe essere "Cammina".
Uomo e cane camminano entrambi ma il loro modo di muoversi è indubbiamente diverso.
Il poliformismo è la capacità di una classe di ridefinire il comportamento "cammina" nelle classi derivate Uomo e Cane, facendo di fatto assumere alle due entità comportamenti diversi. In assenza di questa caratteristica, il metodo "cammina" dovrebbe essere rimosso dalla classe base e spostato in tutte le classi derivate.
Per implementare il concetto di poliformismo in C#, facciamo riferimento alle classi Persona, Agente e Cliente viste nel paragrafo precedente. Supponiamo di voler implementare un metodo nella classe base che consente alle classi derivate di effettuare un ordine.
La metodologia con cui effettuare l'ordine sarà diversa per l'agente ed il cliente.
Il primo passo sarà quindi quello di creare un nuovo metodo nella classe base che possa andar bene per la maggior parte delle classi derivate che andremo a creare (l'esempio è relativo a sole due classi derivate ma si provi a pensare ad un numero maggiore di tali classi):

```
public virtual void CreaOrdine(string Prodotto)
{
 //codice per la creazione dell'ordine
}
```

Il metodo appena creato nella classe base è preceduto dalla parola chiave virtual. Essa permette alle classi derivate di ridefinirlo.
Nelle due classi derivate (Cliente ed Agente) eseguiremo l'override

(la riscrittura) di questo metodo in modo che possa essere personalizzato in base alle esigenze.
Nella classe Cliente, che userà l'implementazione comune del metodo CreaOrdine, faremo in modo che sia richiamato direttamente il metodo della classe base:

```
public override void CreaOrdine(string Prodotto)
{ base.CreaOrdine(Prodotto);
}
```

Sebbene dovessimo creare nella classe derivata un nuovo metodo, non è stato necessario riscriverne l'implementazione.
Nella classe Agente invece, riscriveremo completamente il metodo della classe base in questo modo:

```
public override void CreaOrdine(string Prodotto)
{
double _PrezzoListino;
double _Sconto;
double _PrezzoScontato;
//Codice per il recupero del prezzo del prodotto
_PrezzoScontato = _PrezzoListino - _Sconto;
//Codice per la creazione dell'ordine;
}
```

Se proviamo ad immaginare un software in cui le tipologie di "Persone" sono diverse, è evidente, anche qui, il risparmio di codice introdotto dalla programmazione ad oggetti.

4.5 ASTRAZIONE

L'astrazione è l'ultimo ma importante concetto relativo alla programmazione ad oggetti. Per astrazione si intende la definizione di

una classe il cui scopo è quello di essere usata solo come classe base. Per capire meglio questo concetto, ritorniamo alla nostra classe persona del paragrafo 5.3.
Nella sua definizione abbiamo implementato proprietà, metodi ed un costruttore. La classe persona è quindi, a tutti gli effetti, una classe istanziabile da cui possiamo creare un oggetto persona.
Ma riflettendoci un po', questo potrebbe non avere molto senso.
In un ipotetico software che ha la necessità di gestire clienti e fornitori, l'istanza di un oggetto Persona servirebbe a ben poco se non proprio a causare degli errori.
Ecco quindi che è stato introdotto il concetto di astrazione.
Una classe astratta si definisce usando la parola chiave abstract sia in fase di definizione della classe, sia in fase di definizione dei metodi di cui si vuole fare l'override nelle classi derivate.
Altra particolarità delle classi astratte è che l'implementazione dei metodi che devono essere sottoposti ad override deve essere omessa.
Una classe astratta infatti, deve essere usata come strumento per organizzare al meglio il codice. Altro aspetto importante è che la classe astratta non è istanziabile, sia perché i metodi non prevedono implementazione, sia perché è il concetto alla base di questo tipo di classi che ne impedisce la creazione di un'istanza.
Vediamo quindi come cambia la creazione della nostra struttura di classi vista nel paragrafo precedente alla luce di questa nuova implementazione.

```
abstract class AbstractPersona {
protected string _Nome;
protected string _Cognome;
protected int _Età;
protected string _Indirizzo;
protected string _Telefono;
public abstract string Nome {
get;
```

```
set;
}
public string Cognome {
get { return _Cognome; }
set { _Cognome = value; }
}
public int Età {
get { return _Età; }
set { _Età = value; }
}
public AbstractPersona() {
}
public abstract void CreaOrdine(string Prodotto);
}
}
```

Innanzitutto la classe è stata dichiarata come abstract ed i campi come protected (vedi capitolo 4.1.1). A titolo di esempio, abbiamo impostato ad abstract anche la proprietà Nome. Dichiarandola in questo modo, abbiamo dovuto necessariamente eliminare l'implementazione dei metodi di accessibilità get e set. L'altra modifica è stata fatta al metodo CreaOrdine, anche esso dichiarato come abstract e da cui è stata eliminata l'implementazione.
La classi derivate da una classe abstract, a differenza di quanto visto nel precedente esempio, devono necessariamente implementare tutti i metodi della classe base.
Le due classi Agente e Cliente andranno quindi modificate come segue:

```
class abCliente : AbstractPersona {
private double _Credito;
public override string Nome {
 get {
```

```
  return _Nome;
   }
  set {
   _Nome = value;
  }
  }
 public double Credito {
 get { return _Credito; }
 set { _Credito = value; }
 }
 public abCliente() {
 }
 public override void CreaOrdine(string Prodotto) {
 //Implementazione del metodo di creazione dell'ordine
 }
}
```

e:

```
 class abAgente : AbstractPersona {
 private string _AreaDiCompetenza;
 public override string Nome {
 get {
 return _Nome;
 }
 set {
 _Nome = value;
   }
  }
 public string AreaDiCompetenza {
 get { return _AreaDiCompetenza; }
 set { _AreaDiCompetenza = value; }
 }
```

```
public abAgente() {

  }
 public override void CreaOrdine(string Prodotto) {
 double _PrezzoListino = 0;  //settato a 0 solo per compilazione
 double _Sconto = 0;         //settato a 0 solo per compilazione
 double _PrezzoScontato = 0; //settato a 0 solo per compilazione
 //Codice per il recupero del prezzo del prodotto
 _PrezzoScontato = _PrezzoListino - _Sconto;
 //Codice per la creazione dell'ordine;
 }
 }
```

Il .NET Framework stesso fa largo uso delle classi astratte al fine di organizzarne al meglio tutto il codice. La più importante classe astratta inclusa nel framework è Object.
Nel capitolo 3, avevamo già fatto accenno a questo tipo. Tutte le classi infatti derivano da esso, anche quelle implementate da noi.
Object è infatti incluso implicitamente come classe base anche se non dichiarato con i due punti.

4.6 SEALED

Le classi definite come sealed (sigillate), sono l'esatto opposto delle classi astratte. In pratica, una classe sealed non può essere ereditata e deve essere istanziata.Il motivo principale è quello di impedire che le nostre classi siano estese. All'interno del .NET Framework esistono numerosi oggetti dichiarati come sealed. Si pensi ad esempio al SqlDataAdapter di ADO.net. Questo oggetto permette di gestire la comunicazione tra un altro oggetto denominato DataSet ed il Data Base. Essendo dichiarato come sealed, non sarà possibile ereditarlo ed estenderlo.Operazione che potrebbe causare problemi all'applicazione.

4.7 I NAMESPACE

Nel corso di questo libro, abbiamo già fatto riferimento ai namespace, citando quelli più importanti inclusi nel .NET Framework. Questa entità è estremamente radicata nel framework che, oltre a farne largo uso, viene data la possibilità ai programmatori di creare i propri namespace.In questo capitolo daremo uno sguardo ai namespace cercando di capirne la loro reale utilità. Ci soffermeremo anche sulla creazione dei nostri namespace, su alcune regole di base da seguire e sull'utilizzo di essi nelle nostre applicazioni.

4.8 A COSA SERVONO

Lo scopo principale dei namespace è l'organizzazione del codice. Grazie ad essi infatti, è possibile organizzare le nostre classi in modo gerarchico, aiutandoci a risolvere eventuali conflitti tra i vari identificatori di un programma.Pensiamo ad esempio all'istruzione Console.WriteLine() usata spesso negli esempi di questo libro. Gerarchicamente, WriteLine() è un metodo statico (non abbiamo infatti bisogno di creare un'istanza), dell'oggetto Console. Quest'ultimo si trova all'interno del namespace System. Il percorso completo del metodo WriteLine() è dunque System.Console.WriteLine(). Abbiamo accennato, ed approfondiremo nei prossimi paragrafi, alla possibilità di creare i nostri namespace ed abbiamo anche detto che essi ci aiutano a risolvere i conflitti all'interno delle nostre applicazioni. Supponiamo infatti di avere la necessità di creare un nostro metodo denominato WriteLine() e che esso faccia parte di una nostra classe denominata, guarda caso, Console(). Richiamando questo metodo con Console.WriteLine() è evidente l'ambiguità con il corrispettivo che appartiene al namespace System.Definendo però un nostro namespace, ad esempio MyProgram, l'ambiguità è risolta in quanto i due metodi omonimi, sarebbero richiamati rispettivamente con System.Console.WriteLine() e con MyProgram.Console.WriteLine().

4.9 USING

Dover richiamare ogni volta un elemento, ad esempio, della Base Class Library, specificandone il nome completo, è decisamente scomodo e rende il codice decisamente poco pulito.
Si pensi non tanto a System.Console.WriteLine(), che comunque è abbastanza compatto ma, ad esempio, all'enumerazione System.Security.Cryptography.X509Certificates.X509SubjectKeyIdentifierHashAlgorithm. ShortSha1;
Per rendere più comodo l'uso dei namespace, è stata introdotta la direttiva Using, che fornisce la possibilità di richiamare il metodo senza doverne necessariamente specificare il nome completo.
Nell'esempio dell'enumerazione X509SubjectKeyIdentifierHashAlgorithm, usando correttamente la direttiva Using scriveremo:

```
using System.Security.Cryptography.X509Certificates;
```

e, nel resto del codice, ci basterà richiamarla con:

```
X509SubjectKeyIdentifierHashAlgorithm. ShortSha1;
```

con un evidente risparmio di codice scritto.

4.10 ALIAS

L'alias di un namespace consente di specificare un nome diverso per un namespace. Questa direttiva torna molto utile per risolvere i casi di ambiguità accennati nel paragrafo 6.1.
Per meglio comprendere i vantaggi introdotti da questa direttiva, supponiamo che il nostro metodo personalizzato Console.WriteLine() abbia il seguente nome completo: MyCompany.MyProgram.MyAssembly.Console.WriteLine().
Attraverso la direttiva Using vista nel precedente paragrafo, abbiamo la possibilità di scrivere

```
using MyCompany.MyProgram.MyAssembly;
…
Console.WriteLine("Questa è il metodo della mia classe MyProgram");
```

Ma esiste ancora un problema. Il namespace System deve necessariamente essere inserito in ogni programma.
Ne consegue che il caso di ambiguità continua a persistere in quanto Console.WriteLine() è presente in due namespace diversi.
Grazie alla direttiva alias però, possiamo assegnare un nome diverso a MyCompany.MyProgram.MyAssembly in questo modo:

```
using myNS = MyCompany.MyProgram.MyAssembly;
```

da questo momento, sarà possibile richiamare la nostra Console.WriteLine() usando la forma abbreviata:

```
myNS.Console.WriteLine("Questa è il metodo
della mia classe MyProgram");
```

risolvendo definitivamente il problema dell'ambiguità ed usando allo stesso tempo una forma abbreviata.

4.11 CREAZIONE DI NAMESPACE PERSONALIZZATI

Creare un proprio namespace è un'operazione decisamente semplice: basta usare la parola chiave namespace ed includere tutti i membri di un namespace all'interno delle parentesi graffe.
Un esempio immediato è:

```
namespace ioProgrammo
{
//codice interno al namespace}
```

Attraverso la definizione di questo namespace, sarà possibile fare riferimento ai membri in esso contenuti usando ioProgrammo.membro da qualsiasi parte del programma.Ma dare un nome così semplice e breve, potrebbe generare problemi di ambiguità visti nel paragrafo 6.1. Supponiamo infatti di aver scritto un metodo denominato Console.WriteLine() all'interno del namespace ioProgrammo relativo a questo libro. Poco dopo, scrivendo un articolo sempre per ioProgrammo, definiamo un metodo con lo stesso nome ma con implementazione diversa. All'interno di uno stesso progetto, ci troveremmo nuovamente di fronte ad un caso di ambiguità. E' importante considerare infatti che il namespace è un'entità logica e non fisica, quindi può essere suddiviso su classi e file diversi.
Come risolviamo l'ambiguità?
La soluzione più sensata è quella di usare un namespace maggiormente descrittivo. Nel nostro caso potremmo definire:

```
namespace ioProgrammo.libri.CSharp2.Capitolo6 {
//codice interno al namespace
}
```

Per le applicazioni commerciali, si usa in genere specificare il nome della compagnia, del prodotto ed eventualmente quello del componente.Supponendo di sviluppare un componente per l'accesso ai dati, il namespace potrebbe essere:

```
namespace Mindbox.DataMapper.SqlDataMapper {
//codice interno al namespace
}
```

4.12 STRUTTURE

Le strutture sono un particolare tipo di classi il cui scopo è quello di incapsulare dei campi.

Esse si differenziano dalle classi per un motivo fondamentale: le struct sono Value Type e non Reference Type come le classi. Sebbene una struttura può implementare costruttori, metodi etc., se dovessimo avere bisogno di tali funzionalità è opportuno valutare l'utilizzo di una classe.E' bene sottolineare che una struttura non può ereditare da una classe o da un'altra struttura ma può implementare interfacce.

4.13 DEFINIRE STRUTTURE

La definizione di una struttura è molto simile a quella di una normale classe come visibile nell'esempio seguente:

```
public struct Person {
public string FirstName;
public string LastName;
public int Age;
}
```

Una delle differenze più evidenti rispetto ad una classe è l'assenza, come abbiamo visto nel paragrafo precedente, di costruttori e metodi.

4.14 CREARE STRUTTURE

Una volta creata la struttura, è molto semplice utilizzarla. Innanzitutto non abbiamo la necessità di usare il costruttore new.
Ci basta definire la struct e valorizzarne i campi come visibile nel seguente codice:

```
Person person;
person.FirstName = "Michele";
person.LastName  = "Locuratolo";
person.Age = 29;
```

```
Console.WriteLine("FirstName: {0}\r\nLastName:
{1}\r\nAge:
{2}",
person.FirstName,
person.LastName,
person.Age.ToString()
);
```

Nella prima riga dichiariamo la struttura di tipo Person chiamandola person. Nelle successive righe ne valorizziamo i campi ed infine di mostriamo su una console.

4.15 INTERFACCE

Le interfacce rappresentano un aspetto molto importante della programmazione ad oggetti: esse definiscono infatti i comportamenti che un oggetto deve implementare.
Nel capitoli precedenti, parlando di classi ed oggetti, avevamo usato come riferimento rispetto al mondo reale, un'automobile o una persona. Prendendo come spunto l'oggetto automobile, possiamo dire che esso appartiene ad una famiglia (deriva da), ad esempio, mezzi di trasporto. Tutti i mezzi di trasporto avranno, come è logico immaginare, dei comportamenti comuni come Accelera o Frena. Un'interfaccia è l'elemento, all'interno del nostro software, il quale definisce questi comportamenti che le nostre classi derivate devono necessariamente implementare.
Spesso, per definire un'interfaccia, si usa il termine di contratto. L'interfaccia è, se vogliamo, una sorta di contratto a cui una classe deve attenersi.
La descrizione delle interfacce ricorda molto da vicino quella delle classi astratte viste nel paragrafo 5.5. Ma ci sono delle differenze importanti da tenere presente.
Innanzitutto, C# non supporta l'ereditarietà multipla.

Questo vuol dire che le nostre classi possono derivare al massimo da una sola classe base. Non c'è invece limite all'implementazione di interfacce. Se ci rendiamo conto che una nostra classe dovrebbe derivare da più classi base, possiamo definirne una significativa e spostare il resto del codice (con opportuni accorgimenti) in delle interfacce.
Le altre differenze fondamentali sono le seguenti:

- in una classe astratta, i metodi in essa definiti possono avere un'implementazione. In un'interfaccia invece no.
- Una classe astratta può derivare da altre classi ed interfacce. In un'interfaccia può derivare solo da altre interfacce.
- Le classi astratte possono contenere dei campi mentre le interfacce no.
- Le strutture possono derivare da interfacce ma non da classi astratte

Viste le caratteristiche di base di questo elemento del linguaggio, il modo migliore per comprenderle è quello di utilizzarle. Nei prossimi paragrafi vedremo infatti come definirle ed utilizzarle e scopriremo quali vantaggi concreti esse introducono.

4.16 DEFINIRE ED IMPLEMENTARE INTERFACCE

Per meglio comprendere la definizione e l'uso delle interfacce, riprendiamo l'esempio dell'automobile visto nel capitolo 4. La nostra auto, a questo punto, potrebbe derivare da un'interfaccia in cui definiamo i metodi Accendi, Spegni, Accelera e Frena:

```
public interface IMezzoTrasporto {
void Accendi();
void Spegni();
```

```
void Accelera();
void Frena();
}
```

Se a questo punto provassimo a creare una nostra classe automobile che implementa l'interfaccia IMezzoTrasporto, senza definirne i quattro metodi:

```
public class Automobile : IMezzoTrasporto
{
}
```

otterremmo un errore di compilazione che ci segnala che non abbiamo implementato i membri dell'interfaccia.
Come detto in precedenza, l'interfaccia è un contratto ed i metodi in essa definiti devono essere implementati.
La nostra classe dovrà quindi diventare:

```
public class Automobile : IMezzoTrasporto {
public void Accendi() {
//Implementazione di Accendi()}
public void Spegni() {
//Implementazione di Spegni( }
public void Accelera()
{
//Implementazione di Accelera()
}
public void Frena() {
//Implementazione di Frena()}
}
```

In questo modo, la nostra classe rispetta il contratto imposto dall'interfaccia e può essere compilata.

4.17 SOVRASCRIVERE I MEMBRI DELL'INTERFACCIA

Come detto in precedenza, una classe può ereditare da una sola classe base e da diverse interfacce.
Può capitare che, una nostra ipotetica classe derivata, derivi sia da una classe base che implementa l'interfaccia, sia dall'interfaccia stessa:

```
public class NewAutomobile : Automobile, IMezzoTrasporto {
}
```

In questo caso ovviamente, non ci sarà chiesto esplicitamente dal compilatore di implementare i membri di IMezzoTrasporto in quanto già implementati da Automobile.
Può però capitare che si abbia la necessità di ridefinire il comportamento di un metodo (ad esempio Accelera) nella nostra nuova classe.
Se provassimo a farlo con la sintassi:

```
public void Accelera(){
}
```

otterremmo un avviso dal compilatore il quale ci indica che il nostro nuovo metodo "nasconde" quello della classe base.
Per "sovrascrivere" il membro dell'interfaccia, dobbiamo usare la parola chiave new nella definizione del metodo:

```
public class NewAutomobile : Automobile, IMezzoTrasporto
{
public new void Accelera()
{//Implementazione di Accelera() }}
```

In questo modo, l'avviso del compilatore sparirà.

4.18 IMPLEMENTAZIONE ESPLICITA DELL'INTERFACCIA

Usando le interfacce, non è difficile che si possa incorrere in problemi di ambiguità dei metodi da implementare.
Questo particolare caso è dovuto principalmente al fatto che, in un'applicazione, si possono usare componenti di terze parti o componenti scritti da altri team di sviluppo e che non possono essere modificati.Vediamo un semplice esempio per capire meglio il problema. Supponiamo di avere due interfacce ed una classe che le implementa entrambe:

```
public interface IA {
void MyMethodA();
void MyMethodB();
}
 public interface IB {
 void MyMethodA();
 void MyMethodB();
 void MyMethodC();
}
```

Come evidente, le due interfacce definiscono due metodi con lo stesso nome (MyMethodA e MyMethodB) e l'interfaccia IB aggiunge un ulteriore metodo denominato MyMethodC.
Un'eventuale classe che implementi le due interfacce, potrebbe definirsi completa già in questo modo:

```
 public class MyClass : IA, IB {
 public void MyMethodA() {
 //Implementazione di MyMethodA()
  }
 public void MyMethodB() {
 //Implementazione di MyMethodB()
```

```
}
public void MyMethodC() {
//Implementazione di MyMethodC()
}
```

E' evidente che MyMethodA e MyMethodB fanno riferimento a quelli definiti nell'interfaccia IA e, apparentemente, non c'è modo di implementare i membri aventi lo stesso nome dell'interfaccia IB.
Una soluzione a questo tipo di problematica esiste e si definisce "implementazione esplicita".
Consiste nello specificare esplicitamente l'interfaccia di cui stiamo definendo il membro con il seguente codice:

```
void IB.MyMethodA() {
//Implementazione esplicita di IB.MyMethodA()
}
```

Il nome del metodo da richiamare viene prefissato dall'interfaccia che lo definisce. Ma questa dichiarazione, da sola, non è sufficiente alla corretta esecuzione del codice.
Da parte dell'oggetto che userà la nostra classe MyClass, il metodo implementato esplicitamente non è visibile in quanto privato.
Per accedervi, dobbiamo necessariamente effettuare il casting (vedi capitolo 3) dell'oggetto istanziato all'interfaccia che definisce il metodo in questione.
Tale implementazione è fattibile con il seguente codice:

```
static void Main(string[] args) {
MyClass prova = new MyClass();
prova.MyMethodA();
IB explicitImpl = prova;
explicitImpl.MyMethodA();
}
```

Supponendo di implementare i metodi MyMethodA e IB.MyMethodB con un'istruzione Console.WriteLine() che mostri due messaggi diversi, sulla console vedremo scrivere:

```
Implementazione di MyMethodA()
Implementazione esplicita di IB.MyMethodA()
```

In sostanza, l'istanza della nostra classe MyClass viene castata alla relativa interfaccia (di cui i membri sono di default pubblici) e viene richiamato il relativo metodo che, concretamente, farà riferimento all'implementazione IB.MyMethodA della classe MyClass.

4.19 POLIFORMISMO CON LE INTERFACCE

Il poliformismo implementato con le interface è molto simile a quello implementato con le classi base ed è molto utile in quanto una stessa interfaccia può essere usata in più classi.
Supponiamo di avere un'interfaccia implementata da due classi diverse strutturate in questo modo:

```
 public interface ITest {
 void Message();
}
 public class MyTest1 : ITest {

 public void Message() {
 Console.WriteLine("MyTest1.Message()");
 }
}
  public class MyTest2 :
ITest
{
```

```
public void Message() {
Console.WriteLine("MyTest2.Message()");
  }
}
```

Come evidente, un po' come accade per il poliformismo viso nel capitolo 5, le due classi definiscono un comportamento diverso per lo stesso metodo.

4.20 QUANDO USARE LE INTERFACCE E LE CLASSI ASTRATTE

All'inizio, vista la grande similitudine tra le classi astratte e le interfacce, può non essere semplice decidere quanto usare l'una e quando l'altra.
Esiste però una regola abbastanza semplice che può aiutare nella scelta. Conviene definire un'interfaccia se le funzionalità che vogliamo definire sono usate da un gran numero di oggetti.
In questo caso infatti, oltre ad essere più probabile che il nostro oggetto debba essere costretto a ridefinire il comportamento, è quasi sicuro che gli oggetti possano appartenere a famiglie diverse.
Lo scopo di una classe base infatti è quello di raggruppare degli oggetti correlati. Più l'insieme è vasto e meno è probabile che tra gli elementi ci sia correlazione.

ARRAY, INDICI E COLLECTIONS

Un array è un particolare tipo di struttura utile a memorizzare e gestire una serie di elementi in qualche modo relazionati tra loro. Tale struttura è un tipo riferimento, quindi memorizzata nell'heap.
Il più semplice array è quello di tipo monodimensionale. In esso, tutti gli elementi vengono memorizzati sotto forma di elenco. Spesso questo tipo di array viene chiamato vettore.
Il secondo tipo di array è quello multimensionale. Volendolo rappresentare in qualche modo, esso assomiglierebbe ad una tabella con righe e colonne. Il terzo tipo è l'array jagged. E' sostanzialmente un array multidimensionale ma "frastagliato".
In termini pratici, un array è un contenitore in cui possiamo inserire i nostri elementi che "dovrebbero" essere in qualche modo relazionati. Se pensiamo ad un contenitore di clienti, inserire nello stesso vettore dei prodotti potrebbe non essere la scelta migliore. Gli elementi inseriti in queste strutture sono identificabili da un elemento di tipo int chiamato indicizzatore, sintatticamente identificato da una coppia di parentesi quadre [].
Gli array hanno inoltre due caratteristiche importanti la prima è che l'indice di un array è a base 0. Vuol dire che il primo elemento avrà indice 0, la seconda è che queste strutture hanno dimensione fissa, specificata in fase di creazione (o esplicitamente o implicitamente). Tale limitazione è superabile con altri tipi di collection che, ovviamente, sono più pesanti da gestire. Sebbene nel .NET Framework 2.0 sia presente il metodo generico System.Array.Resize<T>, il lavoro svolto da questo metodo è quello di creare un nuovo array con le nuove dimensioni e copiare al suo interno il contenuto della precedente struttura. Operazione non certo leggerissima specie in presenza di array complessi.
Oltre agli array puri, nel .NET Framework sono presenti una serie di collection utili a vari scopi. Si tratta di Stack, Queue, ArrayList etc. che vedremo, con esempi pratici, nel corso di questo capitolo.
Ultima ma non meno importante considerazione: sebbene siano pre-

senti delle collection predefinite, nulla ci vieta di creare le nostre collection personalizzate.
Per agevolare il lavoro di noi sviluppatori, nel .NET Framework sono incluse un serie di interfacce da implementare nel nostro codice.
Negli ultimi paragrafi di questo capitolo, analizzeremo le interfacce da implementare per creare le nostre collections in base al lavoro che esse dovranno svolgere.
Dopo questa presentazione, non ci resta che toccare con mano queste strutture.

5.1 ARRAY

Gli array sono classi di tipo collezione integrate nel linguaggio CSharp2005. Lo scopo di un Array è quello di memorizzare ed eventualmente manipolare, i dati che in esso saranno memorizzati.
Questo tipo di struttura fornisce un supporto ai dati di tipo matriciale (a matrice).
Esistono, nel .NET Framework, tre tipi di Array: monodimensionale, multidimensionale, jagged.
Nei prossimi paragrafi vedremo come si utilizzano e quando sono utili.

5.1.1 Array Monodimensionali

Un array monodimensionale è un array ad una sola dimensione che fornisce la possibilità di archiviare un elenco di elementi dello stesso tipo. La dichiarazione di un array monodimensionale tipo si effettua con la seguente sintassi:

```
Tipo[] NomeArray [inizializzazione];
```

Un primo esempio di un semplice array monodimensionale è il seguente:

```
string[] MioElenco = {“uno”, “due”, “tre”, “quattro”, “cinque”};
```

Un array può essere inizializzato anche specificando il numero di elementi che esso dovrà contenere, specificando tale valore tra le parentesi quadre come mostrato nel codice seguente:

```
string[] MioElenco = new string[5]
{"uno", "due", "tre", "quattro", "cinque" };
```

In questo caso, se in fase di inizializzazione dovessimo raggiungere il numero massimo di elementi, la compilazione fallirebbe.
L'accesso agli elementi di un array avviene o attraverso i cicli for (capitolo 3.6.3) e foreach (capitolo 3.6.4), o accedendo direttamente all'elemento che ci interessa specificandone l'indice tra le parentesi quadre. Nel primo caso, utilizzando i cicli, dovremmo scrivere il seguente codice:

```
Console.WriteLine("*** Esempio FOR ***");
foreach (string s in MioElenco)
{
Console.WriteLine(s);
}
```

oppure, usando il foreach:

```
 Console.WriteLine("\r\n*** Esempio FOREACH ***");
 for (int i = 0; i < MioElenco.Length; i++) {
 Console.WriteLine(MioElenco[i]);
}
```

Per accedere direttamente all'elemento dell'array invece, andrà specificato l'indice tra le parentesi quadre in questo modo:

```
 Console.WriteLine("\r\n*** Esempio Indicizzatore ***");
 Console.WriteLine(MioElenco[3]);
```

E' importante notare che l'indice degli array è a base 0. Il primo elemento è quindi 0 e non 1.
Il codice sopraripportato, facendo riferimento all'array MioElenco, scriverà sulla console quattro e non tre.

5.1.2 Array Multidimensionali

Gli array multidimensionali sono molto simili a quelli monodimensionali salvo il fatto che, come dice il termine stesso, hanno più di una dimensione. Per capirlo, facciamo un esempio pratico:

```
string[,] Nominativi = new string[,] { { "Michele", "Locuratolo" },
{"Lucia", "Zizzi" }, { "Pinco", "Pallino" } };
```

Come evidente, a differenza dell'array monodimensionale, in questo tipo di array è possibile specificare, per la stessa "riga", più "colonne" (in questo caso due). L'accesso agli elementi dell'array è identico al precedente. Per recuperare infatti il secondo valore del primo elemento, si dovrà scrivere:

```
Console.WriteLine(Nominativi[0,1]);
```

che scriverà sulla console Locuratolo (ricordo che l'indicizzatore è sempre a base 0).
Trattandosi di un array multidimensionale, per iterare su tutti gli elementi in esso contenuti dovremmo usare i cicli for e foreach nel seguente modo:

```
foreach (string s in Nominativi)
{
Console.WriteLine(s);}
```

In questo caso, usando il ciclo for, vedremo visualizzati sulla console tutti gli elementi in sequenza, come fossero parte di un ar-

ray monodimensionale.
Per il ciclo foreach invece, possiamo accedere ciclicamente agli elementi in questo modo:

```
Console.WriteLine("\r\n*** Esempio FOR ***");
for (int i = 0; i < Nominativi.Length/2; i++)
{
Console.WriteLine(Nominativi[i, 0] + " " + Nominativi[i, 1]);
}
```

Innanzitutto, all'interno della definizione del ciclo, dobbiamo considerare che l'array è multidimensionale e che contiene due dimensioni. Ne dobbiamo tenere conto in quanto, la proprietà Nominativi.Lenght conterrà la somma di tutti gli elementi dell'array (nel caso dell'esempio sono sei).Ciclando direttamente usando questo indicizzatore, è evidente che otterremmo un errore in quanto, l'indice più alto del nostro array è 2 (sono in tutto tre elementi). Per accedere poi ai singoli elementi, utilizziamo la seguente sintassi:

```
Console.WriteLine(Nominativi[i, 0] + " " + Nominativi[i, 1]);
```

I nostri elementi della seconda "colonna" infatti, avranno sempre indice 0 o 1. In caso di array multidimensionali di n dimensioni (n colonne), l'indice delle rispettive colonne sarà sempre n-1 a prescindere dalla lunghezza dell'array (numero di righe).

5.1.3 Array Jagged

L'array jagged (frastagliato) è un particolare tipo di array multidimensionale in cui è specificata la dimensione della prima dimensione ma non della seconda.
Vediamo subito un esempio pratico:

```
int[][] jaggedArray = new int[5][];
```

In questa prima riga istanziamo un array di interi specificando la dimensione della prima dimensione.
Procediamo poi all'inserimento dei singoli elementi:

```
jaggedArray[0] = new int[5] { 5, 10, 9, 0, 18 };
jaggedArray[1] = new int[7] { 0, 1, 15, 6, 3, 12, 5 };
jaggedArray[2] = new int[1] { 42 };
jaggedArray[3] = new int[5] { 11, 7, 9, 3, 12 };
jaggedArray[4] = new int[2] { 41, 1 };
```

Andiamo quindi a visualizzare i valori contenuti nell'array con il seguente codice:

```
for (int i = 0; i < 5; i++)
{ for (int j = 0; j < jaggedArray[i].Length; j++)
{
 Console.Write(jaggedArray[i][j].ToString() + ' '); }
 Console.WriteLine();}
```

che stamperà sulla console il seguente output:

```
*** ARRAY JAGGED ***
5 10 9 0 18
0 1 15 6 3 12 5
42
11 7 9 3 12
41 1
```

5.2 FOREACH

Nel paragrafo 3.6.4 avevamo già dato accenno all'istruzione foreach accennando brevemente all'uso, di questa istruzione, nelle collections.

In questo paragrafo, dopo aver compreso cosa è una collection, analizzeremo più in dettaglio questo tipo di ciclo.
L'istruzione foreach permette di iterare tra tutti gli elementi contenuti in una collection.
Foreach significa infatti "per ogni". Tale comando è molto comodo quando abbiamo la necessità di eseguire uno stesso set di istruzioni su tutti gli elementi della nostra collezione. Negli esempi visti sin' ora, all'interno del ciclo foreach abbiamo sempre usato la semplice Console.WriteLine(), utile per mostrare su una console il contenuto della collection. Ma si pensi ad esempio al caso in cui tutti gli elementi devono essere formattati in un certo modo od ancora essere scritti su un Data Base.
La sintassi del ciclo foreach è la seguente:

```
foreach (tipo nella collection)
{
    //istruzione da eseguire }
```

Il modo più immediato per comprendere il modo in cui il costrutto foreach lavora sulle collection è "tradurlo" in linguaggio comune:
"per ogni ……… nella collection ………., fai qualcosa".
E' immediatamente evidente che, per completare la frase, è necessario conoscere due elementi: il tipo di dato contenuto nella nostra collection, e la stessa collection in cui vogliamo eseguire il nostro ciclo. Un esempio tipico è quello già evidenziato nel paragrafo 9.1.1 in cui il ciclo foreach è stato usato per gli elementi della collection MioElenco scrivendo sulla console tutti i valori in essa contenuti.

5.3 INDEXERS

Un indicizzatore è un elemento del linguaggio che ci permette di accedere ai membri di una nostra classe usando una sintassi simile

a quella degli Array.
Essi infatti sono usati internamente dalle classi di tipo collection incluse appunto nel .NET Framework.
Implementare in una classe un nostro indicizzatore è semplice: la sintassi infatti è molto simile alle proprietà viste nel paragrafo 4.7.
Vediamo quindi un semplice esempio pratico di utilizzo di un indicizzatore in una nostra classe.
Supponiamo di avere un oggetto che rappresenti dei prodotti presenti in un ipotetico sito di commercio elettronico.
Il codice di tale oggetto potrebbe essere il seguente:

```
public class Product {
private string _Type;
private string _Name;
private string _Description;
private decimal _UnitPrice;
public string Type {
get { return _Type; }
set { _Type = value; }
}
public string Name {
get { return _Name; }
 set { _Name = value; }
}
public Product() {
}
public Product
(string type, string name, string description, decimal unit_price)
{
Type = type;
Name = name;
Description = description;
UnitPrice = unit_price;
```

```
  }
}
```

Oltre al singolo oggetto, ci sarà sicuramente l'esigenza di avere una lista ordinata di questi Prodotti. In sostanza, abbiamo la necessità di avere una nostra collezione personalizzata di Prodotti.
Abbiamo anche la necessità di accedere sia in scrittura che in lettura agli elementi contenuti nella nostra lista.
Il nostro ipotetico oggetto ProductList potrebbe essere rappresentato dal seguente codice:

```
public class ProductList {
SortedList products;
public ProductList() {
products = new SortedList();
}
public Product this[int index] {
get {
if (index > products.Count) {
eturn (Product)null;
} else {
return (Product)products.GetByIndex(index);
}
}
set {
if (index < 5) {
products[index] = value;
  }
}
}
}
```

L'indicizzatore, come spiegato in precedenza, è implementato in mo-

do molto simile ad una proprietà in cui, nell'istruzione set, implementiamo a titolo di esempio anche un controllo per impedire che vengano aggiunti più di cinque elementi.
Per usare la nostra nuova ProductList, procediamo come segue:

```
Product myNewMouse = new Product
("Mouse", "IntelliMouse", "Mouse laser cordless", 15.0m);
Product myNewKeyboard =
new Product("Keyboard", "IntelliKeyboard",
"Tasiera ergonomica cordless", 35.0m);
```

Creiamo innanzitutto due istanze della diverse dell'oggetto Product che andremo successivamente ad inserire nella nostra ProductList:

```
ProductList pList = new ProductList();
pList[0] = myNewMouse;
pList[1] = myNewKeyboard;
```

Per accedere, ad esempio, al secondo elemento visualizzandone il contenuto su una console dovremo procedere, come se si trattasse di un'ArrayList, in questo modo:

```
Console.WriteLine("Dati recuperati da ProductList[1]");
Console.WriteLine("\tName:        {0}", ((Product)pList[1]).Name);
Console.WriteLine("\tType:        {0}", ((Product)pList[1]).Type);
Console.WriteLine("\tDescription: {0}", ((Product)pList[1]).Description);
Console.WriteLine("\tUnit
Price:  {0}", ((Product)pList[1]).UnitPrice.ToString());
```

Il cui risultato sarà:

```
Dati recuperati da ProductList[1]
Name:       IntelliKeyboard
```

Type:	Keyboard
Description:	Tasiera ergonomica cordless
Unit Price:	35,0

5.4 LE COLLECTION INCLUSE IN SYSTEM.COLLECTION

Gli array puri, come visto nei precedenti paragrafi, sono tipi di strutture molto leggere e comode per gestire delle liste di elementi.
La loro leggerezza e semplicità, se da un lato ne fa il loro punto di forza, dall'altro le rende poco versatili per gestire strutture più complesse ed effettuare operazioni quali il sorting o il ridimensionamento.
Come detto in precedenza, abbiamo la possibilità di creare le nostre collezioni tipizzate che risolvono al meglio le problematiche specifiche ma, come vedremo nel corso di questo capitolo, l'implementazione non è un lavoro immediato.
L'arrivo poi dei generics e delle relative collezioni generiche, rende spesso superflua la creazione di collections tipizzate.
Per venire incontro alle esigenze più comuni, il .NET Framework include già una serie di collections predefinite molto utili al fine di ridurre il carico di lavoro dei programmatori.
Esse si trovano nel namespace System.Collections e possono essere utilizzate per gestire in modo più complesso ed ottimizzato diversi tipi di collections.
Vediamo insieme le più importanti ed il contesto in cui esse si utilizzano.

5.4.1 ArrayList

L'ArrayList è una collection particolare che porta con se due grandi vantaggi. Il primo è che la lista si ridimensiona automaticamente in base alla quantità di elementi che inseriamo al suo interno.
Il secondo vantaggio è che ArrayList accetta in ingresso qualsiasi tipo di dato.Vediamo subito un paio di esempi pratici per comprendere

il contesto di utilizzo di ArrayList.
Definiamo subito un ArrayList con capacità due:

```
ArrayList myList = new ArrayList(2);
Console.WriteLine("Capacità della lista: {0}",
myList.Capacity.ToString());
```

Sulla console verrà quindi scritto:

```
Capacità della lista: 2
```

Iniziamo ad aggiungere degli elementi, stampando sempre a console la capacità della lista:

```
for (int i = 0; i < 10; i++)
{
myList.Add(i);
Console.WriteLine("Capacità della lista: {0}",
myList.Capacity.ToString());
}
```

L'output di questo codice, sulla console, dimostrerà che la capacità di myList si è adeguata al contenuto passando a 2 a 16.
Vediamo ora il secondo vantaggio di questo tipo di collection: la possibilità di accettare più tipi in ingresso.
Definiamo quindi una nuova ArrayList ed inseriamo due tipi diversi al suo interno:

```
ArrayList myList2 = new ArrayList(2);
myList2.Add("Ciao");
myList2.Add(1);
```

Questa operazione è possibile in quanto, il tipo che ArrayList accet-

ta in ingresso è Object e, come sappiamo, tutti i tipi del .NET Framework derivano da esso.
Il seguente codice

```
for (int i = 0; i < myList2.Capacity; i++)
{
Console.WriteLine(myList2[i].ToString()
+ " - " + myList2[i].GetType().ToString());
}
```

scriverà sulla console:

```
Ciao - System.String
1 - System.Int32
```

Questo vantaggio tipico di questa collection, può però diventare uno svantaggio sia in termini prestazionali che in termini di gestione degli errori.
Sul piano puramente prestazionale, lo svantaggio è generato proprio dal tipo che la collection accetta in ingresso. Object infatti è un tipo riferimento mentre string o int dei tipi valore.
Quando inseriamo nell'ArrayList un tipo valore, come nell'esempio visto in precedenza, il .NET Framework esegue un'operazione di conversione da tipo valore a tipo riferimento.
Tale operazione, possibile in quanto, come già sappiamo, tutti i tipi derivano da Object, è chiamata Boxing.
Viceversa, quando accediamo all'elemento contenuto nella nostra collection, esso deve essere riconvertito da tipo riferimento a tipo valore. Operazione chiamata Unboxing.
Il lavoro svolto dal .NET Framework per lo svolgimento di queste operazioni, penalizza indubbiamente le performance della nostra applicazione.
Il secondo svantaggio è, come accennato, legato all'utilizzo di que-

sto tipo di collection all'interno del nostro codice.
Supponiamo ad esempio di avere la necessità di inserire in un'ArrayList una serie di numeri:

```
ArrayList myList2 = new ArrayList();
myList2.Add(1);
myList2.Add(15);
myList2.Add(20);
myList2.Add(6);
```

e, all'interno di un ciclo, effettuarne la somma:

```
int somma = 0;
foreach(int myValue in myList2){
    somma += myValue;
}
Console.WriteLine
("La somma di tutti gli elementi
di myList2 è {0}", somma.ToString());
```

Come è lecito aspettarsi, il risultato dell'operazione sarà 42.
Ma come sappiamo, in myList2 possiamo aggiungere qualsiasi tipo.
Se modificassimo il nostro codice in questo modo:

```
ArrayList myList2 = new ArrayList();
myList2.Add(1);
myList2.Add(15);
myList2.Add(20);
myList2.Add(6);
myList2.Add("Ciao");
```

non avremo nessuna segnalazione in fase di compilazione in quanto l'operazione è lecita, ma avremo un errore a runtime (nel caso

specifico, una InvalidCastException), nel momento in cui si cercherà di effettuare la somma dei valori all'interno del ciclo.
Se nell'esempio riportato è decisamente difficile incappare nel problema, si pensi al fatto che la collezione potrebbe essere popolata a runtime prendendo magari dati da un data Base.
In quel caso, il rischio di inserire elementi non compatibili con il tipo di operazione che si vuole svolgere su di essi, è abbastanza elevato.ArrayList va quindi usata con una certa accortezza o, in alternativa, conviene usare l'utilizzo di una delle collection generiche che vedremo nel capitolo 12.

5.4.2 Hashtable

La collection HashTable rappresentano un elemento estremamente utile in molte applicazioni in quanto molto comoda per memorizzare svariati tipi di informazioni.
In una collection di questo tipo è possibile memorizzare un qualsiasi object a cui viene associata una chiave di tipo stringa. HashTable infatti è strutturata in coppie chiave-valore al fine di rendere estremamente versatile e veloce l'inserimento e l'uso delle entità in essa memorizzate.
Il primo valore della coppia sarà sempre la chiave che sarà utilizzata al posto del classico indicizzatore che abbiamo usato per le collections precedenti.
Vediamone subito un paio di esempi per comprenderne le potenzialità.

```
Hashtable myTable = new Hashtable();
myTable.Add("Michele", "Locuratolo");
myTable.Add("Pinco", "Pallino");
```

Nel codice sopra riportato, innanzitutto creiamo una HashTable chiamata myTable a cui aggiungiamo due elementi.
Se vogliamo recuperare il secondo elemento della prima coppia, lo

facciamo usando la sua chiave:

```
Console.WriteLine
("Il cognome di Michele è {0}",
myTable["Michele"].ToString());
```

Qualora volessimo, come per gli esempi visti nei precedenti paragrafi, visualizzare l'intero contenuto della nostra HashTable, dovremmo far riferimento agli elementi inseriti attraverso un oggetto DictionaryEntry che rappresenta appunto la nostra coppia chiave-valore:

```
 foreach (DictionaryEntry entry in myTable)
{
 Console.WriteLine("Chiave: {0}\r\nValore: {1}",
 entry.Key, entry.Value);
}
```

che scriverà sulla console:

```
 Chiave: Michele
Valore: Locuratolo
 Chiave: Pinco
Valore: Pallino
```

Fin qui il comportamento della HashTable è simile ad un Array multidimensionale come quelli visti nel paragrafo 9.1.1 .
Vediamo quindi un esempio più significativo che chiarisce meglio le potenzialità di questo tipo di collection.
Supponiamo di avere un software al cui interno si gestiscono delle entità di tipo prodotto, riprendendo il codice del paragrafo 9.3
Nella nostra applicazione creiamo tre istanze diverse di questa classe creando, di fatto, tre oggetti diversi:

```
Product myMouse = new Product("Mouse", "IntelliMouse",
"Mouse laser cordless", 15.0m);
Product myKeyboard = new Product("Keyboard",
"IntelliKeyboard", "Tasiera ergonomica cordless", 35.0m);
Product myDVD = new Product("DVD", "DVDRW",
"Masterizzatore DVD +/- R", 45.0m);
```

Ora non ci resta che creare la nostra lista di product che può essere eventualmente passata ad un altro elemento della nostra applicazione:

```
Hashtable myProduct = new Hashtable();
myProduct.Add(myMouse.Name, myMouse);
myProduct.Add(myKeyboard.Name, myKeyboard);
myProduct.Add(myDVD.Name, myDVD);
```

Come evidente dal codice, la nostra HashTable denominata myProduct ha come chiave il nome del prodotto mente come valore il relativo oggetto di tipo Product.
Vediamo come visualizzare un elenco completo degli elementi contenuti nella nostra collection:

```
foreach (DictionaryEntry entry in myProduct)
{ Console.WriteLine(entry.Key);
Console.WriteLine("\tType:        {0}",
((Product)entry.Value).Type);
Console.WriteLine("\tDescription: {0}",
((Product)entry.Value).Description);
Console.WriteLine("\tUnitPrice:
{0}", ((Product)entry.Value).UnitPrice.ToString());}
```

che scriverà sulla console il seguente testo:

```
IntelliMouse
```

```
Type:        Mouse
Description: Mouse laser cordless
Unit Price:  15,0
IntelliKeyboard
Type:        Keyboard
Description: Tasiera ergonomica cordless
Unit Price:  35,0 DVDRW
Type:        DVD
Description: Masterizzatore DVD +/- R
Unit Price:  45,0
```

Se avessimo la necessità di visualizzare un singolo elemento, ad esempio il dettaglio dell'intellimouse, dovremmo scrivere:

```
Console.WriteLine("\tType:       {0}",
((Product)myProduct["IntelliMouse"]).Type);
```

Avere una collection i cui elementi sono costituiti da una coppia chiave-valore, fornisce numerosi vantaggi in termini di leggibilità ed usabilità del codice.
Le HashTable hanno numerose applicazioni pratiche.
Data la loro particolare struttura, stà al programmatore individuare, all'interno del proprio codice, il modo migliore per sfruttarne tutti i vantaggi.

5.5 COLLECTION INTERFACES

Tutte le collection analizzate fino ad ora hanno in comune una serie di elementi che fanno sì che esse appartengano, alla fine, ad una stessa famiglia. Questa appartenenza è garantita da una serie di interfacce a cui una collection deve adeguarsi al fine di essere considerata tale.
Le interfacce che entrano in gioco per la definizione di una collection sono le seguenti:

Nome Interfaccia	Descrizione
ICollection	Fornisce enumerazione, sincronizzazione e dimensione della collezione
IComparer	Permette di confrontare due oggetti interni alla collezione (utile per l'ordinamento)
IDictionary	Consente la creazione di collection di tipo chiave-valore (come HashTable, SortedList)
IDictionaryEnumerator	Fornisce l'enumeratore per IDictionary
IEnumerable	Ottiene un enumeratore per la collezione
IEnumerator	Permette di ciclare nelle collection con foreach
IHashCodeProvider	Ottiene un codice Hash
IList	Permette la creazione di una collection indicizzata

Tabella 3: le interfacce per la definizione delle collections.

Ogni collection, per definirsi tale, deve implementare almeno una delle interfacce riportate nella tabella 3.
Le interfacce da implementare devono essere scelte in base al tipo di collezione che abbiamo la necessità di creare.
Per una semplice collection, potrebbe bastare l'implementazione di ICollection. Ovviamente, se abbiamo la necessità di creare un'ArrayList, ICollection non sarà sufficiente.
Attraverso l'implementazione di queste interfacce sarà quindi possibile implementare le nostre collection personalizzate. Si rimanda alla documentazione disponibile on line l'approfondimento di questo argomento.

5.6 GESTIONE DELLE ECCEZIONI

Scrivere u programma perfetto e privo di errori è il sogno di ogni programmatore e di ogni azienda che opera in questo settore.
Purtroppo però, il concetto di software perfetto è e resterà un sogno.

Da sviluppatori, quello che possiamo fare è sforzarci di progettare e realizzare software con il minor numero di errori possibili e fare in modo che quelli presenti, siano opportunamente gestiti.
In questo capitolo ci limiteremo a studiare quali sono le tecniche per gestire gli errori che il nostro programma può generare una volta consegnato al cliente finale.
A tale scopo, il .NET Framework ha al suo interno un potente sistema per la gestione delle eccezioni. Sistema che, come ormai siamo abituati, possiamo estendere e personalizzare a nostro piacimento essendo basato anche esso su classi ed interfacce.
Per eccezione si intende appunto un evento non previsto che può causare un blocco della nostra applicazione. Il sistema per la gestione delle eccezioni del .NET Framework si basa su un costrutto composto dai comandi try (prova), catch (prendi) e finally (alla fine).
Vedremo nei prossimi paragrafi come gestire questi eventi, quando conviene usare la gestione delle eccezioni e quali sono sia i vantaggi che gli svantaggi.

5.7 SOLLEVARE E GESTIRE LE ECCEZIONI

Quando all'interno di un programma si verifica un evento inaspettato che causa un errore, il CLR "solleva" un'eccezione.
Tale eccezione, se non gestita, provoca un errore che viene mostrato all'utente attraverso un messaggio abbastanza dettagliato, utile ad individuare la causa del problema.
Se l'utente è lo sviluppatore del software, tale errore è sicuramente utile a comprendere le cause che lo hanno generato ed intervenire di conseguenza.
Se invece l'errore viene mostrato all'utente, sarà di poco aiuto e farà calare la fiducia che l'utente ha nella nostra applicazione.
Un tipico errore è quello mostrato in (figura 4).
Tale errore è causato da questa semplice istruzione:

```
int myVar = int.Parse(tbxNumero.Text);
```

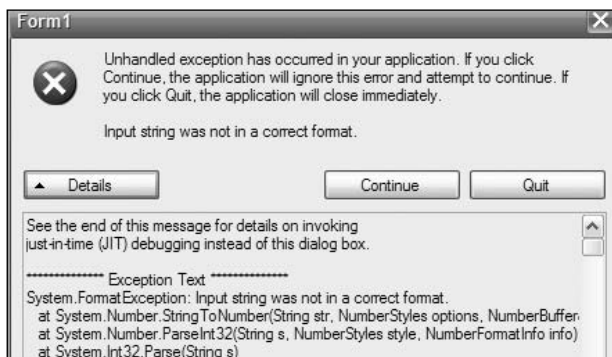



Figura 4: Un tipico errore non gestito dall'applicazione.

in cui, nella text box tbxNumero è stato inserito un testo, ovviamente non associabile ad una variabile di tipo int.
Come evidente dall'immagine, l'errore, sebbene descrittivo, è poco "user friendly" e non spiega chiaramente all'utente cosa ha causato l'errore.
La nostra applicazione, in un caso come quello mostrato nell'esempio, dovrebbe quantomeno mostrare ad un utente normale la causa del problema e, se esso non è bloccante, dare la possibilità di ripetere l'operazione.
Vediamo ora cosa cambia con l'utilizzo di un costrutto try/catch.
Innanzitutto dobbiamo individuare quale istruzione del nostro codice potrebbe generare un'eccezione.
In questo caso è molto semplice, ma non sempre abbiamo questa fortuna.
Il secondo passo è quello di inserire l'istruzione (o il seti di istruzioni) nell'apposito costrutto in cui, nella prima parte (try), inseriamo appunto l'istruzione, mentre nella seconda (catch), inseriamo il codice che dovrà gestire l'eventuale errore generato dal nostro codice.
Il nostro esempio si trasforma quindi in:

```
try
{
myInt = int.Parse(tbxNumero.Text);
}
catch (Exception ex)
{
MessageBox.Show
("Si è verificato un errore nell'applicazione!
\r\nL'errore è
"+ex.Message, "Eccezioni",
MessageBoxButtons.
OK,
MessageBoxIcon.Error);
}
```

In questo modo, al verificarsi dell'eccezione, mostriamo all'utente un messaggio che descrive in modo chiaro e comprensibile ai non addetti ai lavori l'errore che si è verificato (figura5).
La gestione delle eccezioni è molto importante in quanto permette, all'interno del blocco catch, di eseguire qualsiasi tipo di operazione, permettendoci di prevenire blocchi totali del programma e, cosa molto importante, ad eseguire un log degli errori.

5.8 GESTIONE DI PIÙ ECCEZIONI

Nelle nostre applicazioni, possono verificarsi più eccezioni diverse che devono essere gestite, magari in modi diversi.
Si pensi ad esempio ad un software che ha la necessità di scrivere dei dati su un Data Base remoto.
Al momento della scrittura, può verificarsi un errore che può dipendere da varie cause.
Potrebbe essere assente una connessione di rete, o il server su cui risiede il nostro Data Base potrebbe non essere raggiungibile.

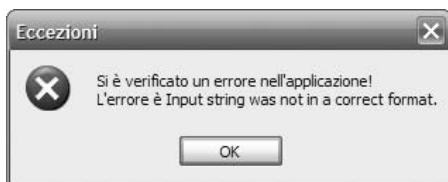



Figura 5: Lo stesso errore ma gestito con un costrutto try/catch

Altri tipi di eccezioni che potrebbero sorgere possono nascere da svariate problematiche come l'assenza di permessi opportuni etc.
E' evidente che, per ognuna di queste problematiche si avrà la necessità di comportarsi in maniera diversa. Fortunatamente, nel .NET Framework, ogni eccezione è di un tipo ben specifico. Nell'esempio visto nel paragrafo precedente, ad esempio, l' eccezione sollevata è di tipo FormatException in quanto cercavamo di convertire in int una stringa passata attraverso una textBox. Specificando il tipo di eccezione nel costrutto catch, abbiamo la possibilità di filtrare in modo dettagliato le eccezioni da gestire e, all'occorrenza, scrivere del codice opportuno per gestirle.Ad esempio, riprendendo il codice visto in precedenza, possiamo mostrare all'utente un messaggio di errore più dettagliato:

```
try
{myInt = int.Parse(tbxNumero.Text);}
catch (FormatException ex)
{MessageBox.Show("Si è verificato un errore
nell'applicazione!\r\nImmettere un numero
nella TextBox", "Eccezioni",
MessageBoxButtons.OK,
MessageBoxIcon.Error); }
```

Che restituirà il seguente messaggio:

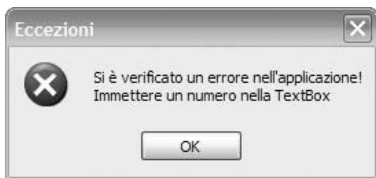



Figura 6: Lo stesso errore ma gestito con un costrutto try/catch e dettagliata.

Ma se si verifica un'eccezione di tipo diverso? Ad esempio inseriamo un numero troppo grande per essere compreso nel range degli int? L'eccezione cambia di tipologia passando da una FormatException ad una OverflowException che il nostro codice, nel blocco catch, non sarà più in grado di gestire in quanto abbiamo chiaramente specificato quale eccezione gestire.Il .NET Framework ci da comunque la possibilità di aggiungere blocchi catch aggiuntivi rendendo possibile la diversa gestione degli errori in base alla loro tipologia.
Se trasformiamo il codice di sopra in questo modo:

```
try {
myInt = int.Parse(tbxNumero.Text);}
catch (FormatException ex) {
MessageBox.Show("Si è verificato un errore
nell'applicazione!\r\nImmettere un numero nella TextBox",
"Eccezioni", MessageBoxButtons.OK, MessageBoxIcon.Error);}
catch (OverflowException oex)
{MessageBox.Show
("Si è verificato un errore nell'applicazione!
\r\nIl numero immesso non è compreso nel range degli int",
"Eccezioni", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

otterremo il seguente messaggio in caso di OverflowException:

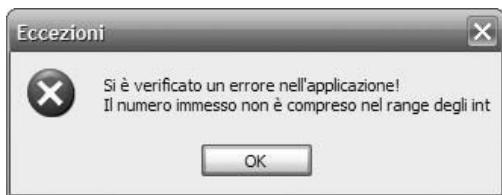



Figura 7: La gestione dell'eccezione di tipo overflow.

mentre continueremo ad avere il messaggio di (figura 7) per la FormatException. Potremmo continuare per tutti i tipi di eccezioni che potrebbero essere sollevati dal nostro oggetto specifico, ma non è raro che ci sia o un'eccezione che non abbiamo considerato oppure una che poco ha a che fare con l'oggetto specifico ma viene sollevata per altre cause.
Per nostra fortuna, il sistema di gestione delle eccezioni è gerarchico. Questo stà a significare che le nostre eccezioni sono gestite dal blocco catch partendo dalla più dettagliata fino ad arrivare a quella meno dettagliata.
L'eccezione più generica di tutte è Exception che deve essere inserita nell'ultimo blocco catch dell'elenco. In questo modo, sarà l'ultima ad essere gestita e ci da la possibilità, anche in casi estremi, di mostrare un messaggio più semplice all'utente e di gestire a nostro piacimento gli eventi da compiere (come log, riavvio dell'applicazione etc.).

5.9 FINALLY

Il blocco finally, sebbene sia opzionale, riveste grande importanza nella gestione delle eccezioni. In esso infatti, vanno inserite tutte le operazioni che devono essere comunque svolte anche in caso di crash dell'applicazione. Per meglio comprendere il blocco finally, facciamo un esempio concreto.

Prendiamo ad esempio il seguente codice:

```
System.IO.StreamWriter sw = new
System.IO.StreamWriter("logEccezione.txt", true);
string[] myString = new string[5];
for (int i = 0; i < 8; i++) {
myString[i] = i.ToString();
sw.WriteLine(i.ToString());}
sw.Close();
```

Esso solleverà un'eccezione dovuta al fatto che abbiamo definito un vettore di cinque elementi, ma, nel ciclo for, cerchiamo di inserirne otto. L'eccezione sarà di tipo IndexOutOfRangeException. Quello che ci interessa ai fini dell'esempio però è lo StreamWriter.
Grazie ad esso infatti, scriveremo su un file di testo gli stessi elementi che inseriamo nel vettore grazie all'istruzione sw.WriteLine(i.ToString()). Il problema nasce dal fatto che, per far sì che il file sia correttamente scritto, l'istruzione sw.Close() deve essere necessariamente eseguita. Cosa che nel nostro codice non avviene a causa dell'eccezione.Gestendo però il ciclo for all'interno di un costrutto try/catch ed inserendo l'istruzione sw.Close() all'interno del blocco finally, avremo la certezza che questa istruzione sarà sempre eseguita, tanto in caso di eccezione quanto durante la normale esecuzione del nostro software.
Vediamo quindi come modificare il codice:

```
private void TestMultipleException()
{System.IO.StreamWriter sw = new
System.IO.StreamWriter("logEccezione.txt", true);
string[] myString = new string[5];
try {
for (int i = 0; i < 8; i++) {
myString[i] = i.ToString();
```

```
sw.WriteLine(i.ToString());}
} catch (IndexOutOfRangeException iex) {
MessageBox.Show("Stai cercando di inserire troppi elementi nell'array",
 "Eccezioni", MessageBoxButtons.OK, MessageBoxIcon.Error);
} catch (Exception ex) {
MessageBox.Show("Di seguito una descrizione dell'errore:\r\n" +
ex.Message,
 "Eccezioni", MessageBoxButtons.OK, MessageBoxIcon.Error);
} finally {
sw.WriteLine("Chiusura");
sw.Close();}
}
```

Sebbene verrà visualizzato il messaggio relativo all'eccezione del tipo IndexOutOfRangeException, il file di testo sarà correttamente popolato e chiuso con la dicitura "Chiusura". Il blocco finally torna molto utile quando abbiamo a che fare con risorse che corrono il rischio di restare bloccate, impedendo ad altri utenti o altre parti del software di impegnarle. Se inoltre, all'interno del blocco try lavoriamo con un oggetto che implementa il pattern Dispose (visto nel paragrafo 4.4.2), è buona norma richiamarlo nel finally.

5.10 ECCEZIONI PERSONALIZZATE

Come abbiamo visto nel precedente paragrafo, le eccezioni sono gerarchiche e, per gestirle correttamente, dobbiamo iniziare con quella più dettagliata per risalire, mano mano verso la più generica Exception.Per quanto il .NET Framework sia ricco di eccezioni gestibili, potrebbero non coprire tutte quelle di cui abbiamo bisogno.
Per questo, grazie al meccanismo dell'ereditarietà visto nel paragrafo 5.3, possiamo definire le nostre eccezioni personalizzate.
Una semplice eccezione personalizzata potrebbe essere definita in questo modo:

```
class MyPersonalException :
Exception {
public MyPersonalException()
: base("Eccezione personalizzata") {
}
```

in cui il costruttore della nuova eccezione richiama direttamente quello della classe base passandole un messaggio predefinito.

5.11 RISOLLEVARE UN'ECCEZIONE

Le eccezioni, molto spesso, non vengono gestite nel punto preciso del codice in cui si presentano.
Pensiamo ad esempio ad un'applicazione complessa strutturata in più livelli in cui esiste un componente per l'accesso al Data Base.
Non è difficile che lo stesso componente sia usato in tipologie diverse di applicazioni come programmi per Windows, servizi o pagine web.
E' chiaro che sarebbe impensabile, stando agli esempi che abbiamo fatto fino ad ora, mostrare una message box in quel contesto. Chiaramente, il codice che genera l'eccezione, sarà stato chiamato da altre parti del codice come ad esempio l'interfaccia utente, magari seguendo vari passaggi attraverso i vari livelli di cui è composto il nostro programma.Nel .NET Framework, le eccezioni si propagano all'interno del nostro codice finché non viene trovato un gestore specifico per l'eccezione (o il più generico Exception).
Qualora questo gestore non dovesse essere trovato, ci troveremo di fronte un messaggio come quello visto nel paragrafo 10.1.
Il meccanismo con cui le eccezioni si propagano viene spesso definito "bubling".
Come una bolla infatti, l'eccezione risale tutti i livelli dell'applicazione fino alla superficie che spesso è l'interfaccia utente.
Se ne deduce che non è necessario gestire un'eccezione nel punto spe-

cifico in cui viene sollevata ma solo al livello più alto.
Questo però non è sempre vero.
Ritornando all'esempio del Data Base, potrebbe ad esempio essere utile compiere determinate operazioni nel blocco catch per liberare delle risorse o scrivere su un log.
In questo modo però, l'eccezione risulterebbe già gestita e non arriverebbe mai al livello più alto dell'applicazione.
Per ovviare a questo inconveniente, esiste l'istruzione throw che ci da la possibilità di sollevare nuovamente l'eccezione.
Questa istruzione torna molto utile anche per trasformare un'eccezione in un tipo diverso.
Per capirlo, completiamo l'esempio del paragrafo precedente facendo in modo che la nostra nuova eccezione venga sollevata e gestita.Innanzitutto dobbiamo provvedere a gestire l'eccezione di base che potrebbe essere sollevata dall'istruzione che andiamo ad eseguire.
Nel blocco catch poi, risolleviamo l'eccezione trasformandola in questo modo:

```
private void testMethod() {
try {
myInt = int.Parse(tbxNumero.Text);
} catch (Exception ex) {
throw new MyPersonalException();
}
}
```

Per gestirla, come abbiamo visto negli esempi precedenti, non ci basterà scrivere il seguente codice:

```
private void btnCustomException_Click(object sender, EventArgs e) {
try {
testMethod();
```

```
} catch (MyPersonalException myEx) {
MessageBox.Show("Si è verificato un errore nell'applicazione!\r\n
Di seguito una descrizione dell'errore:\r\n" + myEx.Message,
"Eccezioni", MessageBoxButtons.OK, MessageBoxIcon.Error);
  }
}
```

5.12 UTILIZZARE CORRETTAMENTE LE ECCEZIONI

Le eccezioni sono uno strumento molto comodo da utilizzare nei nostri software. Tanta comodità però ha un prezzo: le prestazioni. Sollevare e propagare un'eccezione non è un'operazione "a costo zero", vale quindi la pena spendere qualche parola per capire meglio quando conviene usarle e quando no.La regola è dettata dalla stessa parola: "eccezione". Il senso è proprio quello di indicare un evento non previsto, eccezionale appunto. Premesso ciò, è importante ragionare in questi termini: se è possibile prevedere un determinato evento, gestirlo con del codice apposito anziché con un'eccezione.Un esempio molto esplicativo potrebbe essere relativo ad una procedura di LogIn. Si sarebbe tentati di gestire un'eccezione personalizzata (ad esempio InvalidUser) per gestire il LogIn fallito. Ma questo comportamento è facilmente prevedibile: se l'utente sbaglia l'inserimento dei dati richiesti, la procedura non gli permetterà di accedere all'applicazione. In questo caso specifico, la scelta più sensata è indubbiamente quella dell'utilizzo di un costrutto if/else.Questa regola è da tenere sempre presente quando si decide di usare le eccezioni. Pena il calo di prestazioni delle nostre applicazioni.

5.13 DELEGATI ED EVENTI

Il .NET Framework contiene dei costrutti denominati delegati ed eventi. Il loro utilizzo è molto importante in quanto consentono di ef-

fettuare delle "chiamate a collegamento preposto" definite tecnicamente "in late binding" molto utili per procedure di callback o semplicemente per richiamare dei metodi.
Le operazioni in late binding hanno la particolarità di verificarsi a runtime (in fase di esecuzione del nostro programma) anziché in fase di compilazione.
Tutte le chiamate ai metodi viste fin'ora infatti, sono ben definite in fase di compilazione attraverso appositi riferimenti nel codice compilato. La situazione è diversa in questo tipo di costrutti.In questo capitolo vedremo i delegati e gli eventi, analizzandone sia il loro modo di funzionamento sia facendo degli esempi reali di utilizzo.

5.14 DELEGATES

I delegati sono elementi del .NET Framework che permettono di richiamare dinamicamente diversi metodi in fase di esecuzione del programma.
Un delegato, concettualmente, è molto simile ad un puntatore a funzione, concetto noto ai programmatori C++.
Il vantaggio dei delegati rispetto ai vecchi puntatori consiste nel fatto di essere type-safe. Un delegato infatti ha una firma (come i metodi) ed un tipo di ritorno ben definito. Questo riduce notevolmente il rischio di bug derivanti dalla natura non type-safe dei vecchi sistemi.La sintassi per definire un delegato è la seguente:[modificatore] delegate TipoDiRitorno NomeDelegate([parametri])in cui è ben evidente sia la necessità di specificare un tipo di ritorno, sia la possibilità di specificare i parametri da utilizzare per utilizzare il delegato.

5.14.1 Definizione dei delegati

La sintassi definita nel precedente paragrafo però, non è sufficiente per la gestione corretta di un delegato.
Per creare ed usare questi elementi infatti, c'è la necessità di seguire alcuni passi fondamentali:

- il primo passo è la definizione del delegato usando la sintassi appena spiegata, come ad esempio:

```
public delegate void SimpleDelegate(string value);
```

- dopo aver definito il delegato, bisogna crearne un'istanza usando la parola chiave new. Nell'istanza del delegato appena definito, bisogna specificare il metodo che esso dovrà richiamare in fase di esecuzione:

```
SimpleDelegate del = new SimpleDelegate(MyTest);
```

Il metodo da richiamare dovrà avere la stessa firma e lo stesso tipo di ritorno del delegato.

- si definisce il metodo che dovrà essere usato a runtime dal delegato:

```
public void MyTest(string value)
{
MessageBox.Show(value);
}
```

- l'ultimo passo è quello di usare (invocare) il delegato appena definito con il seguente codice:

```
del("test");
```

oppure

```
del.Invoke("test");
```

Una sintesi del codice completo usato per definire i passi appena

descritti è il seguente:

```
namespace DelegatiEdEventi
{
public delegate void SimpleDelegate(string value);
public partial class Form1 : Form {
public Form1() {
InitializeComponent();
}
public void MyTest(string value) {
MessageBox.Show(value);
}
private void btnExecuteMyDelegate_Click
(object sender, EventArgs e) {
SimpleDelegate del = new SimpleDelegate(MyTest);
del("test");
del.Invoke("test");
 }
}
```

L'esempio è volutamente strutturato in modo molto semplice. In esso infatti, sarebbe stato più sensato richiamare direttamente il metodo MyTest, dal click sul bottone definito dal metodo btnExecuteMyDelegate_Click anziché richiamarlo usando il delegato.
Nel prossimo paragrafo vedremo quali sono i reali vantaggi dell'utilizzo di questi elementi.

5.14.2 Uso dei delegati

Vediamo quindi qual è il vantaggio di usare i delegati sfruttando come base l'esempio mostrato nel paragrafo precedente.
Supponiamo che, nel nostro codice, si abbia la necessità monitorare l'esecuzione della nostra applicazione nella finestra di debug di Visual Studio 2005 al fine di verificare il corretto funzionamento.

Con le nozioni appese sin'ora, realizzeremmo un semplice metodo che, sfruttando una funzionalità inclusa nel namespace System.Diagnostics, ci permetta di assolvere alla problematica.
Con l'utilizzo dell'applicazione, ci rendiamo conto che sarebbe necessario memorizzare le stesse informazioni anche nel registro eventi di Windows.
La prima cosa che faremmo è includere tale funzionalità nel metodo Log creato in precedenza.
Può però capitare che non sia possibile o sia del tutto sconveniente modificare la classe che effettua il log della nostra applicazione. Una possibile soluzione, in fase di progettazione, è proprio quella di sfruttare i delegati.
La potenzialità di questo mezzo risiede proprio nel fatto che il metodo di destinazione del delegato (metodo target), non viene richiamato direttamente dal nostro codice ma appunto attraverso il delegato.
Il nostro codice andrà strutturato in questo modo: innanzitutto definiamo il delegato con la sintassi vista in precedenza, definendo quindi il tipo di ritorno e la firma che il metodo dovrà implementare.

```
public delegate void LogDelegate(string message);
```

Scriviamo poi la nostra classe che si occuperà di effettuare il log della nostra applicazione.
Dato che abbiamo la necessità di implementare più di un metodo che rifletta la destinazione delle nostre informazioni da loggare, strutturiamo il metodo DoLog in modo che accetti un target specifico ed ovviamente il messaggio da scrivere nel log.
Il metodo target sarà quello che verrà richiamato a runtime per eseguire l'operazione concreta.

```
public delegate void LogDelegate(string message);
```

```
class Log
{
public void DoLog(LogDelegate target, string message)
{
StringBuilder sBuilder = new StringBuilder();
sBuilder.Append("[");
sBuilder.Append(DateTime.Now.ToString());
sBuilder.Append(message);
target.Invoke(sBuilder.ToString());
}
}
}
```

Sebbene non si sia ancora definito il metodo che realmente eseguirà il log, possiamo tranquillamente compilare la nostra soluzione senza ottenere nessun errore.
Ora non ci resta che implementare il log vero e proprio.
Iniziamo dall'inserimento del messaggio nella finestra di debug di Visual Studio 2005 creando una nuova classe che chiameremo Logger:

```
class Logger
{
/// <summary>
public static void DebugLog(string message)
{
System.Diagnostics.Debug.Write(message);
}
   }
}
```

Il metodo DebugLog accetta in ingresso una stringa, rispettando la firma del delegato LogDelegate.

Nella nostra applicazione, ci basterà creare un'istanza della classe Log e richiamare il metodo DoLog a cui, come visibile dal codice descritto, dovremo passare il messaggio da loggare ed il metodo di destinazione che eseguirà il log concreto:

```
Log myLog = new Log();
myLog.DoLog(Logger.DebugLog,
"Click su bottone");
```

Il risultato sarà quello in figura 8.

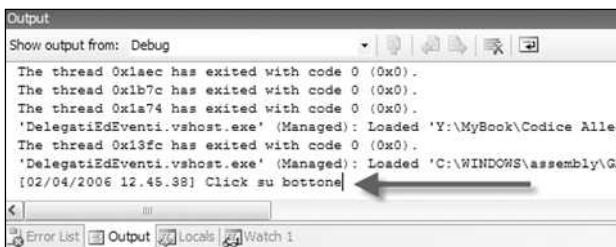



Figura 8: Il delegato eseguito che scrive nella finestra debug di Visual Studio 2005

Fin qui non c'è nessuna novità di rilievo, ma come abbiamo specificato all'inizio, vogliamo implementare anche un sistema per loggare lo stesso messaggio anche nel visualizzatore eventi di Windows, senza dover modificare la classe Log.Grazie all'uso dei delegati, sarà sufficiente implementare il metodo necessario ad eseguire il log anche nel registro eventi direttamente nella classe Logger in questo modo:

```
public static void EventLog
(string message)
{
System.Diagnostics.EventLog.WriteEntry
```

```
("Esempi Libro", message,
System.Diagnostics.
EventLogEntryType.Information);
}
```

che richiameremo grazie al delegate, aggiungendo questa semplice riga alla nostra applicazione, subito sotto quella vista in precedenza per richiamare DebugLog:

```
myLog.DoLog(Logger.EventLog, "Click su bottone");
```

Il risultato sarà quello in figura 9.
Come evidente dal codice mostrato, il nostro software richiamerà sempre il metodo DoLog in cui verrà specificato il metodo che dovrà essere eseguito.
Sarà il nostro delegato poi ad indirizzare la chiamata al metodo concreto.

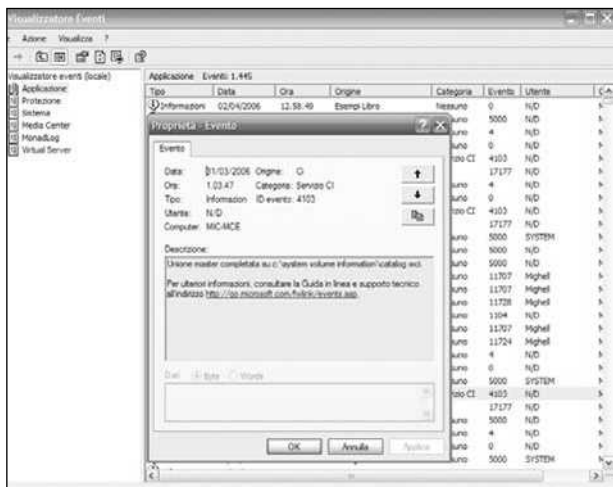



Figura 9: Un Web Server e i programmi CGI-BIN

5.14.3 Chiamate asincrone ai delegates

I delegati possono essere richiamati in modo sincrono (come abbiamo visto sin'ora) o in modo asincrono. La chiamata asincrona ad un delegato, fa sì che le operazioni da esso eseguite vengano svolte in un thread separato da quello che ha eseguito la chiamata.
Il vantaggio di questa particolare modalità di esecuzione risiede nel fatto che, operazioni complesse, possono impiegare diverso tempo per essere eseguite. Se richiamate nel thread principale, l'applicazione resterà bloccata fino alla fine dell'esecuzione, rendendo l'applicazione poco piacevole da utilizzare nonché sprecando tempo che potrebbe essere usato per svolgere altre operazioni.
Addentrarsi nella programmazione denominata multithreading non è cosa semplice e bisogna conoscere alcuni aspetti relativi al funzionamento dei processori.
Cercheremo però di rendere semplice il concetto con un facile esempio.
Analizziamo il seguente codice:

```
public delegate int MyTestAsyncDelegate
(out System.DateTime Start, out System.DateTime End);
class Program {
static void Main(string[] args)
{
 MyTestAsyncDelegate testDelegate = MyTestMethod;
 System.DateTime Start;
 System.DateTime End;
 IAsyncResult asyncRes = testDelegate.BeginInvoke
 (out Start, out End, null, null);
 Console.WriteLine
("Il delegate è stato richiamato in un thread separato nell'istante {0}.",
 DateTime.Now.ToLongTimeString());
 Console.WriteLine
 ("Esecuzione di altre operazioni nel thread principale.");
```

```
Console.WriteLine
("Il metodo MyTestMethod è ancora in esecuzione nel secondo thread.");
Thread.Sleep(1000); //sospendo il thread principale per 1 secondo
 Console.WriteLine
("In questo momento sono le {0}", DateTime.Now.ToLongTimeString());
 while (!asyncRes.IsCompleted) {
  Console.Write(".");
   Thread.Sleep(500); //Usato solo per rallentare la scrittura dei punti
  }
 int TimeElapsed = testDelegate.EndInvoke(out Start, out End, asyncRes);
  Console.WriteLine
 ("\r\nEsecuzione iniziata nell'istante {0}", Start.ToLongTimeString());
 Console.WriteLine
("Esecuzione terminata nell'istante {0}", End.ToLongTimeString());
 Console.WriteLine("Tempo trascorso: {0}",
TimeElapsed.ToString());
 }
 public static int MyTestMethod
 (out System.DateTime Start, out System.DateTime End)
{
 Start = DateTime.Now;
 Thread.Sleep(5000);
//Sospendo l'esecuzione per 5 secondi
End = DateTime.Now;
 int TimeElapsed = (End - Start).
 Seconds; //Tempo necessario all'esecuzione
 return TimeElapsed;
  }
  }
}
```

Notiamo subito un paio di cose familiari come la dichiarazione di un delegato:

```
public delegate int MyTestAsyncDelegate(out System.DateTime Start,
 out System.DateTime End);
```

ed un metodo che ne rispetta la firma:

```
ublic static int MyTestMethod(out System.DateTime Start,
 out System.DateTime End)
{…
}
```

All'interno di questo metodo, eseguiamo quattro semplici operazioni:

- valorizziamo il parametro Start
- "congeliamo" l'esecuzione per cinque secondi
- valorizziamo il parametro End
- calcoliamo il tempo trascorso che sappiamo essere di 5 secondi

Se invocassimo il delegato con il classico sistema, il flusso di esecuzione della nostra applicazione si sposterebbe nel metodo MyTestMethod e ne attenderebbe la conclusione, senza permettere altre operazioni.
Se invece richiamiamo il delegato con il metodo BeginInvoke(), il metodo target viene eseguito in un thread secondario, permettendoci di svolgere altre operazioni. A riprova di quanto affermato, se mandiamo in esecuzione il codice dell'esempio, vedremo sulla console il seguente output:

```
Il delegate è stato richiamato in un thread separato nell'istante 19.10.18.
 Esecuzione di altre operazioni nel thread principale.
 Il metodo MyTestMethod è ancora in esecuzione nel secondo thread.
 In questo momento sono le 19.10.19
```

```
Esecuzione iniziata nell'istante 19.10.18
Esecuzione terminata nell'istante 19.10.23
Tempo trascorso: 5
Premere un tasto per continuare . . .
```

Come si può notare dagli orari visualizzati sulla console, il delegato è stato richiamato alle 19.10.18 e, durante la sua esecuzione, è stato possibile continuare a mostrare dei messaggi sulla console.
Altra piccola particolarità: il metodo eseguito nel thread secondario impiega cinque secondi per essere eseguito.
Attraverso il ciclo

```
while (!asyncRes.IsCompleted)
{
 Console.Write(".");
 Thread.Sleep(500);
//Usato solo per rallentare la scrittura dei punti
}
```

scriviamo sulla console un punto ogni 500 millisecondi.
Dato che l'esecuzione del metodo MyTestMethod impiega 5000 millisecondi, ci aspettiamo di contare dieci punti.
Se contiamo quelli visualizzati sulla console, ci accorgeremo che sono nove.
Come mai?
Nel thread principale, abbiamo inserito la seguente istruzione:

```
Thread.Sleep(1000); //sospendo il thread principale per 1 secondo
```

che sospende l'esecuzione di un secondo.
Questo piccolo accorgimento dimostra che il secondo è stato perso solo dal thread principale (quello relativo alla console), mentre il thread secondario ha continuato la sua esecuzione.

5.15 EVENTI

Un evento, nel .NET Framework, è un avvenimento che un oggetto ha la necessità di notificare ad altri oggetti. Un esempio immediato per comprendere gli eventi è pensare ad un pulsante nella nostra applicazione: quando si clicca su di esso, avviene qualcosa nel nostro programma come ad esempio l'apertura di un nuovo form o la scrittura dei dati in un Data Base. Il bottone ha semplicemente notificato all'applicazione che è stato cliccato. In un altro punto dell'applicazione, il click effettuato sul bottone è stato intercettato ed è stato eseguito del codice.
Il .NET framework fa largo uso degli eventi, ma questo non esclude che si possano realizzare eventi personalizzati sulle classi del nostro software, come ad esempio l'intercettazione dei dati ricevuti da una porta seriale etc.
L'architettura degli eventi si basa su un sistema di publisher (pubblicatore) ed uno o più subscriber (sottoscrittore). Il publisher è l'oggetto da cui parte l'evento. Il subscriber si "registra" all'evento del publisher per intercettarlo attraverso un delegato il cui metodo target è quello che risponderà all'evento intercettato.
Per lo stesso evento generato dal publisher è possibile registrare anche più di un subscriber in modo da poter gestire lo stesso evento in più di una classe.

5.15.1 Architettura degli eventi

Per comprendere meglio l'architettura publisher/subscriber relativa agli eventi, si pensi ad esempio ad un'applicazione di scambio messaggi, come ad esempio un programma di chat.
Quanto uno degli utenti registrati al servizio invia un messaggio, questo viene ricevuto da un server che, per tutti gli altri client registrati, diventa un publisher. Il messaggio appena ricevuto dovrà quindi essere inviato a tutti i client che stanno partecipando a quella conversazione specifica: sono i subscriber.
Questo tipo di architettura è tanto semplice quanto versatile.

Tramite gli eventi infatti, un'applicazione, o più probabilmente una parte di essa, potrà comunicare con le altre parti sfruttando il sistema degli eventi.Gli eventuali sottoscrittori potranno essere facilmente aggiunti a runtime e, qualora non avessero più necessità di ricevere le notifiche, potranno essere rimossi dall'elenco dei sottoscrittori.Un esempio molto utile per comprendere meglio questo meccanismo è relativo alla gestione dei parametri di configurazione di un'applicazione.Nel .NET Framework 1.1, soprattutto in ambiente Windows Forms, era molto sentita l'esigenza di notificare all'applicazione, eventuali modifiche ai parametri di configurazione normalmente memorizzati nel file App.config. Tali settaggi infatti, venivano caricati all'avvio dell'applicazione e, un eventuale modifica a runtime, non veniva comunicata all'applicazione finché la stessa non veniva riavviata.Questo limite, sebbene superato già con la precedente versione del .NET Framework attraverso dei componenti esterni, è stato definitivamente superato con il .NET Framework 2.0 sfruttando appunto gli eventi.Senza scendere troppo nei dettagli della nuova architettura, il nuovo meccanismo basa il suo funzionamento su quattro particolari eventi: PropertyChangedEventHandler, SettingChangingEventHandler, SettingsLoadedEventHandler, SettingsSavingEventHandler. Modificando e salvando le impostazioni di configurazione ora, a differenza di quanto accadeva prima, vengono sollevati questi specifici eventi che la nostra applicazione può sottoscrivere per modificare il suo comportamento.Nei prossimi paragrafi vedremo come creare i nostri eventi personalizzati e sottoscrivere ad essi le nostre applicazioni.

5.15.2 Definizione di un evento

La sintassi per la definizione di un semplice evento è la seguente:

```
[accessibilità] event NomeDelegato NomeEvento
```

Come si evince dalla sintassi, gli eventi fanno uso dei delegati per ge-

stire le notifiche.
Il relativo delegato deve quindi essere già definito prima della definizione dell'evento.
Un esempio concreto di definizione di un evento è il seguente:

```
public delegate void ListaEventHandler();
…
public event ListaEventHandler OnAdd;
```

in cui, alla prima riga viene definito il delegato mentre sulla successive viene definito l'evento vero e proprio che chiameremo OnAdd. Ovviamente il nome dell'evento può essere scelto a piacimento ma si consiglia di dargli un nome significativo, che identifichi, all'interno del nostro codice, che si tratta di un evento.
Il .NET Framework contiene già un delegato predefinito per la gestione della maggior parte degli eventi.
Tale delegato si chiama EventHandler ed ha due parametri in ingresso. Il primo parametro rappresenta l'oggetto che scatena l'evento (Object) mentre il secondo, derivato da EventArgs serve a fornire dei parametri opzionali per la gestione dell'evento stesso.
Se ad esempio osserviamo il tipico gestore di un evento click su un bottone, vedremo che esso è strutturato con la seguente sintassi:

```
private void btnShowAll_Click(object sender, EventArgs e) {}
```

in cui sender rappresenta il bottone premuto ed e gli argomenti opzionali.Registrarsi ad un evento è un'operazione molto semplice. Basterà infatti specificare al delegato qual è il metodo preposto alla gestione dell'evento con la seguente sintassi:

```
oggetto.NomeEvento += new NomeDelegato(NomeMetodo);
```

L'operatore += registra a tutti gli effetti il subscriber all'evento pas-

sando al delegato il nome del metodo che gestirà l'evento.
Se volessimo cancellare la registrazione effettuata, dovremmo usare l'operatore -= in questo modo:

```
oggetto.NomeEvento -= new NomeDelegato(NomeMetodo);
```

Dato che gestire gli eventi sugli elementi inclusi nel .NET Framework è molto semplice in quanto è già presente tutta la relativa struttura di gestione, vediamo un esempio pratico di creazione e gestione di un evento su un nostro oggetto che definiremo nel nostro codice.

5.15.3 Definire e gestire i propri eventi

Nel precedente paragrafo abbiamo dato uno sguardo alla definizione tipica di un evento e alla modalità con cui registrarsi o annullare la registrazione all'evento stesso.
In questo paragrafo vedremo come creare un evento personalizzato in grado di notificare una avvenimento ben definito al resto dell'applicazione. Vedremo anche come gestirlo nel nostro codice. Il codice che verrà mostrato, sebbene correttamente funzionante, va inteso come esempio.
Supponiamo di avere la necessità, all'interno del nostro programma, di creare un oggetto che contenga al suo interno una semplice lista (Arraylist) i cui elementi verranno aggiunti da un apposito form (sul piano pratico, la lista potrebbe essere ad esempio popolata da un lettore di codice a barre).
Vogliamo inoltre che, all'inserimento di un elemento in questa lista, in un altro form della nostra applicazione sia notificato questo inserimento.
Sebbene potremmo "giocare" con il metodo che effettua l'inserimento, la soluzione migliore è quella di dotare la nostra lista della capacità di notificare ai suoi sottoscrittori l'avvenuto inserimento.
Creiamo quindi la nostra classe MyList definendone il delegato e l'evento:

```
namespace DelegatiEdEventi {
  public delegate void ListaEventHandler();
  public class MyList {
   public event ListaEventHandler OnAdd;
  public ArrayList Lista = new ArrayList();
  public void Add(string value) {
  Lista.Add(value);
  OnAdd();
    }
   }
 }
```

Il codice è molto semplice: alla riga 2 definiamo il delegato e, dopo la dichiarazione della classe (MyList) alla riga 4 definiamo il nostro evento che sarà di tipo ListaEventHandler (il delegato creato alla riga 2) di nome OnAdd.
Dopo aver istanziato una semplice ArrayList alla riga 5, definiamo un semplice metodo il cui scopo sarà quello di aggiungere una stringa alla lista appena istanziata (riga 7). Subito dopo l'operazione di inserimento, facciamo partire l'evento (firing) alla riga 8 semplicemente richiamandolo (OnAdd).
Il nostro oggetto è ora pronto ad aggiungere stringhe in una ArrayList ed a notificarne l'inserimento a tutti i sottoscrittori.
Creiamo ora il form che si occuperà di inserire i dati. Tale form avrà una textBox in cui inserire il testo da inviare alla lista ed un bottone che effettuerà l'inserimento.
Gestiamo l'evento click del bottone in questo modo:

```
 private void btnCaricaLista_Click(object sender, EventArgs e)
{
 _lista.Add(textBoxValore.Text);
  textBoxValore.Text = string.Empty;
 }
```

dove _lista sarà un'istanza della nostra MyList.
A questo punto abbiamo la lista ed il form che ne inserisce i valori. Manca il subscriber che gestirà l'evento generato dalla nostra lista. Realizziamo quindi un nuovo form con una textBox per la visualizzazione dei valori e gestiamo l'evento OnAdd della nostra lista in questo modo:

```
private void EventSubscriber1_
Load(object sender, EventArgs e)
{
_Lista.OnAdd += new ListaEventHandler
(lista_OnAdd);
}
```

Ancora una volta, _Lista sarà un'istanza della MyList. Tramite l'operatore +=, registriamo il metodo lista_OnAdd al delegato ListaEventHandler visto in precedenza e lo implementiamo come segue:

```
void lista_OnAdd() {
textBoxDati.AppendText
("Nuovo elemento aggiunto: ");
textBoxDati.AppendText
(_Lista.Lista[_Lista.Lista.Count-
1].ToString()+"\r\n");

}
```

Il nostro subscriber è pronto a ricevere le notifiche dalla MyList. Quando eseguiamo il programma, attraverso il primo form richiamiamo la funzione Add di MyList in cui è presente l'istruzione per far partire l'evento (OnAdd).
Questa richiamerà il delegato ListaEventHandler a cui il subscriber si è registrato con l'operatore += . Il delegato, come abbiamo vi-

sto nel paragrafo relativo ai delegati, richiamerà il metodo opportuno (lista_OnAdd) che gestirà l'evento.
Se mandiamo in esecuzione il codice avremo il risultato in figura 10.
Se per qualche motivo volessimo sospendere la gestione dell'evento OnAdd, ci basterà eliminare la sottoscrizione con il seguente codice:

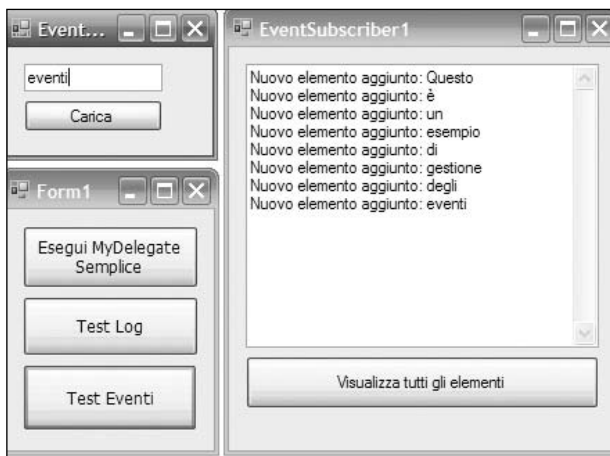



Figura 10: Il risultato del nostro esempio sugli eventi.

```
_Lista.OnAdd -= new ListaEventHandler(lista_OnAdd);
```

L'evento OnAdd verrà comunque sollevato dall'applicazione all'inserimento di nuovi elementi all'interno della lista ma, non essendoci un gestore, non accadrà nulla e la nostra applicazione continuerà a funzionare senza alcun problema.

NOVITÀ DI C# 2.0

Il .NET Framework è giunto alla versione 2.0 e, come è giusto che sia, oltre ad alcune migliorie sono state introdotte alcune nuove funzionalità.
Fermo restando che tutti i concetti fin qui appresi valgono sia per la versione 1 che per la versione 2.0, da questo momento in poi quello che vedremo è applicabile solo alla nuova release, con l'utilizzo di Visual Studio 2005 (o di una delle versioni Express della stessa famiglia).

6.1 GENERICS

Con il termine generics, si identificano questi nuovi elementi che permettono di utilizzare lo stesso codice su istanze di tipi diversi.
Ma cosa significa? Come abbiamo fatto nei capitoli precedenti, prendiamo un esempio che faccia riferimento al mondo reale. Città, è un concetto generico. Una Città non ha caratteristiche particolari se non quella di essere un aggregato urbano dotato di alcune caratteristiche di base (strade, abitazioni etc.). Bari invece, ad esempio, è un concetto ben definito. E' una città (riferimento ad un concetto generico, ma ha delle caratteristiche definite come la localizzazione geografica, il numero di abitanti, le strade etc. Bari, se vogliamo, è una specializzazione di Città. Nulla di famigliare?
In effetti, il concetto appena espresso può essere considerato come un comportamento polimorfico di un'ipotetica classe Città.
Come abbiamo visto nel Capitolo 5.5 relativo al poliformismo, e più in generale in tutto il capitolo 5, avremmo potuto creare una classe base astratta e, grazie al comportamento polimorfico degli oggetti, specializzarla in x classi città diverse.
Ma questa soluzione non è sempre applicabile o, in alcuni casi, sarebbe sconveniente da usare.
Prendiamo subito un esempio, molto semplice, per inquadrare meglio il problema.Supponiamo di voler creare una semplice funzione che scambi i valori di due variabili:

```
private static void Scambia(ref int a, ref int b) {
int appoggio = a;
a = b;
b = appoggio;
}
```

Il funzionamento del metodo Scambia è abbastanza banale: prendiamo in ingresso due valori di tipo int (a e b) passati come riferimento al metodo e ne scambiamo il valore. Approntiamo quindi il nostro codice in modo che siano richieste le due variabili e lo eseguiamo:

```
static void Main(string[] args) {
Console.Write("Immetti il valore di A (int): ");
int a = int.Parse(Console.ReadLine());
Console.Write("Immetti il valore di B (int): ");
int b = int.Parse(Console.ReadLine());
Console.WriteLine("Prima dello scambio: ");
Console.WriteLine("\tA = {0}\r\n\tB = {1}", a.ToString(), b.ToString());
Scambia(ref a, ref b);
Console.WriteLine("Dopo lo scambio: ");
Console.WriteLine("\tA = {0}\r\n\tB = {1}", a.ToString(), b.ToString());
}
```

Come è facile immaginare, il risultato sarà il seguente:

```
Immetti il valore di A (int): 1
Immetti il valore di B (int): 999
Prima dello scambio:
A = 1
B = 999
Dopo lo scambio:
A = 999
B = 1
```

Supponiamo però che nel nostro software ci sia la necessità di scambiare anche due stringhe. Quello che dobbiamo fare, con le nozioni apprese fino ad ora, è semplicemente creare un overload del metodo Scambia che accetta in ingresso due tipi string anziché due int, modificando il nostro codice come segue:

```
private static void Scambia(ref string a, ref string b) {
string appoggio = a;
a = b;
b = appoggio;
}
```

Il risultato dell'operazione sarà identico a quello visto nell'esempio precedente. Ma se ci venisse chiesto di fare la stessa cosa per i tipi long? E per i tipi byte? Finché stiamo lavorando al nostro software, prima di distribuirlo, possiamo aggiungere tutti gli overload che di cui abbiamo bisogno, ma una volta distribuito il software, chi ci assicura che abbiamo implementato tutto? Potremmo anche decidere di implementare il metodo Sambia per tutti i tipi supportati, ma è evidente che questa scelta comporterebbe un lavoro non indifferente e poco logico. Ci toccherebbe infatti scrivere una gran quantità di codice e, comunque, non si potrà conoscere a priori quanti tipi dobbiamo scambiare (il software potrebbe evolvere). I più attenti potrebbero fare questa osservazione: come visto nel capitolo 3.1.1, tutti i tipi derivano da Object. Nulla ci vieta quindi di creare un metodo unico che scambi due Object in modo da avere un metodo unico che funzioni per tutti i tipi. Sebbene la cosa sia fattibile:

```
private static void Scambia(ref Object a, ref Object b)
{ Object appoggio = a;
a = b;
b = appoggio;
}
```

Non è la strada migliore da seguire per due semplici motivi. Il primo è legato a problemi di Boxing/Unboxing. Il tipo di base deve essere prima convertito in oggetto (quindi Reference Type) con un'operazione di Boxing e poi riconvertito nel tipo di destinazione (quindi Value Type). Operazione non certo leggera che introduce ovviamente un calo di prestazioni.Il secondo problema è la perdita del Type Safety.
Se il nostro metodo accetta Object, come nel codice precedente, al suo interno potremmo inserire qualsiasi elemento.
Questo comportamento, sebbene voluto, può causare degli errori a runtime. Supponiamo infatti che il risultato del nostro metodo Scambia, venga poi usato da una funzione che esegue la somma dei valori scambiati. Se il metodo accetta Object e, per un problema qualsiasi, al metodo viene passata una stringa, è evidente che la nostra applicazione solleverà un'eccezione.

6.1.1 Primo approccio con i Generics

Un tipo generico è semplicemente un tipo che viene specializzato automaticamente dal Common Language Runtime sulla base del sistema con cui viene richiamato.
L'esempio è abbastanza semplice ed immediato, non richiederà che un minuto. Vediamo immediatamente un esempio concreto, relativo al nostro metodo Scambia:

```
private static void Scambia<T>(ref T a, ref T b) {
T appoggio = a;
a = b;
b = appoggio;}
```

La prima cosa evidente è la presenza, subito dopo il nome del metodo, di un elemento racchiuso tra parentesi angolari: <T>.
E' una sorta di marcatore, denominato type parameter. Esso dovrà essere passato al metodo nel momento in cui viene richiamato. Come evidente dal codice infatti, i tipi accettati come argomento

del metodo sono T , in riferimento appunto al type parameter passato. Il tipo di dato usato dal nostro novo metodo Scambia, sarà quindi scelto solo a runtime sulla base di come richiameremo il metodo stesso:

```
Scambia<string>(ref a, ref b);
```

In questo caso specifico, il metodo Scambia userà come tipo da gestire una stringa ed accetterà solo quel tipo di dato.
Nulla ci vieta però di richiamare lo stesso metodo con un type parameter diverso:

```
Scambia<int>(ref x, ref y);
```

In quest'altro caso invece, lo stesso metodo verrà specializzato con un intero.Ovviamente, come type parameter, possiamo passare un qualsiasi tipo, come ad esempio il Product usato nell'esempio scritto al paragrafo 9.4.3. Questo primo e semplice esempio, già dimostra i vantaggi introdotti dai generics. Il più evidente è il risparmio di codice. Semplicemente usando un type parameters, abbiamo reso il nostro metodo compatibile con qualsiasi tipo di dato.Il secondo evidente vantaggio è l'aver mantenuto la caratteristica di Type Safety del metodo (con tutti i vantaggi che ne derivano).Richiamando infatti il nostro Scambia con il seguente codice:

```
Scambia<int>(ref x, ref y);
```

non sarà di fatto possibile passare un tipo di dato che non sia int.

6.1.2 Un esempio concreto

Con il termine Stack, si fa riferimento in genere ad una struttura il cui tipo di accesso dei dati è LIFO (Last In, First Out – Ultimo ad entrare, primo ad uscire). Vediamo prima di tutto come avremmo dovuto

realizzare il nostro oggetto senza Generics.
Come è ovvio, per prima cosa avremmo dovuto decidere il tipo di dato che l'oggetto Stack avrebbe dovuto gestire.
Valgono le stesse considerazioni relative al type safety già viste nel precedente paragrafo. Supponiamo quindi di dover gestire degli int:

```
class StackInt {
public Node first = null;
public int top = 0;

public bool Empty {
get { return (first == null); }
  }
public int Pop() {
if (first == null)
throw new Exception("Stack Vuoto.");
else {
--top;
int temp = first.Value;
first = first.Next;
return temp;
   }
  }
public void Push(int value)
{
 ++top;
first = new Node(value, first);
}
public class Node {
public Node Next;
public int Value;
public Node(int value) : this(value, null) { }
public Node(int value, Node next) {
```

```
    Next = next;
    Value = value;
    }
   }
  }
}
```

Come per l'esempio del paragrafo precedente, questo oggetto può essere usato solo con tipi int. Per l'inserimento dei dati nel nostro StackInt, usiamo il metodo Push():

```
StackInt sInt = new StackInt();
sInt.Push(10);
sInt.Push(5);
```

il che ci permette, in esecuzione, di riprendere i valori di cui abbiamo fatto il Push secondo la logica LIFO attraverso il comando Pop(). Essendo uno stack specializzato sul tipo int, il tipo restituito dal metodo Pop() sarà già definito:

```
int i1 = sInt.Pop(); // sarà uguale a 5
int i2 = sInt.Pop(); // sarà uguale a 10
```

come per il paragrafo precedente, supponiamo di avere la necessità di creare uno Stack per la gestione delle stringhe. Dovremmo creare un oggetto diverso. Nel caso analogo all'esempio precedente, usando Object potremmo incappare in un errore a runtime, dovuto alla perdita di type safety. Creando uno Stack di Object
il compilatore ci permetterà di fare una cosa simile a:

```
StackObj sObj = new StackObj();
sObj.Push(10);
sObj.Push("Ciao");
```

ma a runtime, la seguente operazione fallirà:

```
int i1 = sInt.Pop(); // Errore
int i2 = sInt.Pop(); // sarà uguale a 10
```

in quanto l'ultimo valore inserito (il primo ad essere recuperato) è una stringa e non potrà essere convertita in un int.
Nelle collection già presenti all'interno del .NET Framework, come già accennato nel paragrafo 9.4.4, esiste già una collection non generica di tipo Stack che, come tipo di dato in ingresso, accetta appunto Object. Vediamo ora il caso di uno Stack generico.
Sebbene all'interno del namespace System.Collection.Generics esiste già un oggetto Stack generico, ne vediamo un'implementazione personalizzata per compararla con quella dell'esempio precedente relativa a StackInt.

```
class Stack<T> {
public Node first = null;
public int top = 0;
public bool Empty {
get { return (first == null); }
 }
public T Pop() {
if (first == null)
throw new Exception("Stack Vuoto.");
else {
 --top;
T temp = first.Value;
first = first.Next;
return temp;
}
}
public void Push(T value) {
```

```
++top;
first = new Node(value, first);
 }
public class Node {
public Node Next;
public T Value;
public Node(T value) : this(value, null) { }
public Node(T value, Node next) {
Next = next;
Value = value;
}
}
}
```

La prima differenza rispetto al metodo Scambia visto nel paragrafo 13.1.1, è che il type parameter <T> è dichiarato a livello di classe e non di metodo. Significa che, il tipo T che sarà passato al momento della creazione dell'istanza della classe, sarà usato per tutti i metodi presenti nella classe stessa. L'utilizzo di questo nuovo oggetto Stack è molto semplice:

```
Stack<int> sInt = new Stack<int>();
Console.WriteLine("*** Esempio int ***");
Console.WriteLine("Inserimento dati (Push)");
for (int i = 0; i < 10; i++) {
sInt.Push(i);
Console.WriteLine("Push({0})", i.ToString());
}
Console.WriteLine("Elementi contenuti: {0}", sInt.top.ToString());
Console.WriteLine("Il primo valore è {0}", sInt.first.Value.ToString());
Console.WriteLine("Estrazione dei dati da sommare");
int pop1 = sInt.Pop();
int pop2 = sInt.Pop();
```

```
int Somma = pop1+pop2;
Console.WriteLine("Somma dei primi 2 = {0}",
Somma.ToString());
Console.WriteLine("Elementi contenuti: {0}", sInt.top.ToString());
Console.WriteLine("Il primo valore è {0}", sInt.first.Value.ToString());
Stack<string> sString = new Stack<string>();
Console.WriteLine("\r\n*** Esempio String ***");
Console.WriteLine("Inserimento dati (Push)");
for (int i = 0; i < 10; i++)
{ sString.Push("Valore"+i.ToString());
Console.WriteLine("Push({0})", "Valore"+i.ToString());
}
Console.WriteLine
("Elementi contenuti: {0}",
sString.top.ToString());
Console.WriteLine("Il primo valore è {0}",
sString.first.Value.ToString());
Console.WriteLine("Estrazione dei dati (Pop)");
 Console.WriteLine("Primo valore estratto: {0}", sString.Pop());
Console.WriteLine("Secondo valore estratto: {0}", sString.Pop());
Console.WriteLine("Elementi contenuti: {0}", sString.top.ToString());
Console.WriteLine("Il primo valore è {0}", sString.first.Value.ToString());
```

Nella riga 1, creiamo un'istanza dello Stack generico specializzandola con un tipo int. Il compilatore ci permetterà quindi di scrivere:

```
sInt.Push(10);
```

e, per riprendere i valori, ci consentirà di fare:

```
int pop1 = sInt.Pop();
```

ma ci vieterà di scrivere:

```
sInt.Push("Ciao");
```

Abbiamo mantenuto una caratteristica importante: il type safety.
Alla riga 18 inoltre, creiamo un'istanza dello stesso oggetto Stack specializzandola con un tipo String.
Il comportamento sarà analogo al precedente. In esecuzione, il risultato sarà:

```
*** Esempio int ***
 Inserimento dati (Push)
 Push(0)
 Push(1)
 Push(8)
 Push(9)
 Elementi contenuti: 10
 Il primo valore è 9
 Estrazione dei dati da sommare
 Somma dei primi 2 = 17
 Elementi contenuti: 8
 Il primo valore è 7
*** Esempio String ***
 Inserimento dati (Push)
 Push(Valore0)
 Push(Valore1)
 Push(Valore8)
 Push(Valore9)
 Elementi contenuti: 10
 Il primo valore è Valore9
 Estrazione dei dati (Pop)
 Primo valore estratto: Valore9
 Secondo valore estratto: Valore8
 Elementi contenuti: 8
 Il primo valore è Valore7
```

```
Premere un tasto per continuare . . .
```

6.2 TIPI COMPOSTI

Come abbiamo visto nei vari esempi di questo libro, un metodo classico può accettare diversi tipi di parametri in ingresso.
Lo stesso risultato si può comodamente ottenere con i generics, utilizzando più di un type parameter nella dichiarazione del metodo o della classe.
Per farlo, è sufficiente separare tali parametri, all'interno delle parentesi angolari < e > con una virgola, come segue

```
class MyClass<K, V> {
public void MyMethod(K a, V b)
{   //Implementazione }
}
```

La classe MyClass potrà essere quindi istanziata come segue

```
MyClass<string, int> = new MyClass<string, int>();
```

Anche se i type parameter sono stati definiti con lettere diverse (K e V), nulla impedisce di istanziare la classe passando lo stesso tipo al costruttore. Il seguente codice ad esempio compila e funziona senza alcun problema:

```
MyClass<string, string> = new MyClass<string, string>();
```

6.2.1 Overload

Usando i generics, è emersa la possibilità di specializzare il tipo di dato sia al livello di classe che di singolo metodo. Sui metodi però, potrebbe nascere un piccolo paradosso. Facciamo un piccolo passo indietro. Nel capitolo 4.6, abbiamo analizzato una caratteristica mol-

to comoda dei metodo: l’overload. Abbiamo visto che è possibile, semplicemente modificando il numero o il tipo di parametri accettati dal metodo, crearne delle versioni diverse.
Ogni versione viene poi scelta automaticamente dal runtime in base al tipo di parametri passati al metodo.
Premesso ciò, torniamo ai generics. Non bisogna dimenticare che un metodo generico è, in fondo, un classico metodo con la sola differenza che il tipo di dato gestito viene specificato a runtime.
E’ quindi evidente che, anche per questi particolari metodi, deve essere possibile effettuarne l’overload:

```
class MyTest<K, V> {
public void MyTestA(K a) {
//Implementazione del metodo
}

public void MyTestA(V a)
{
//Implementazione del metodo
}
}
```

Se provassimo a compilare il codice di MyTest, la compilazione andrebbe giustamente a buon fine. Allora dov’è il problema?
L’overload di un metodo è possibile solo se il tipo o il numero dei parametri cambia. Nell’esempio, K e V possono essere qualsiasi cosa. Istanziando la classe con due tipi string ad esempio, ci troveremmo di fronte ad un caso di ambiguità.
Il compilatore risolve in parte il problema dell’ambiguità scegliendo il tipo corretto se i due tipi con cui abbiamo costruito l’oggetto sono diversi. Il seguente codice infatti non genera alcun errore:

```
MyTest<string, int> test = new MyTest<string, int>();
```

```
test.MyTestA("Ciao");
test.MyTestA(10);
```

Ma come sappiamo, il nostro oggetto MyTest può essere istanziato con un qualsiasi tipo in ingresso come:

```
MyTest<int, int> test2 = new MyTest<int, int>();
```

In questo caso, la seguente chiamata:

```
test2.MyTestA(10);
test2.MyTestA(20);
```

genererà in compilazione un errore di ambiguità, impedendo di fatto la compilazione. La possibilità di effettuare overload di metodi generici va utilizzata con estrema cautela, specie da chi sviluppa componenti utilizzabili da terze parti. Testare infatti tutte le combinazioni di tipi, non è una cosa semplice.
Il problema dell'ambiguità però, non si presenta se un metodo generico fa l'overload di un metodo non generico. In questo caso specifico, in caso di potenziale ambiguità, verrà richiamato il metodo non generico. In questo caso:

```
class MyNewTest<K, V> {
public void MyTestA(int a)
{
//Implementazione del metodo
}
public void MyTestA(V a)
{
//Implementazione del metodo}
   }
}
```

il codice:

```
MyNewTest<int, int> test2 = new MyNewTest<int, int>();
test2.MyTestA(10);
test2.MyTestA(20);
```

sarà compilato correttamente.
Un sistema per limitare il problema dell'ambiguità (e non solo), consiste nell'utilizzare i Constraints (vincoli) che vedremo nel prossimo paragrafo.

6.2.2 Constraints

Se sviluppiamo un componente da distribuire, il cui codice sorgente non viene reso disponibile, i generics possono introdurre errori.
Se creiamo un oggetto generico con un metodo che scriva sulla console i valori contenuti in una collections, è quantomeno indispensabile che il tipo che verrà passato al metodo implementi l'interfaccia IEnumerable. Se provassimo a scrivere il seguente codice infatti:

```
class MyClass<T> {
public void ElencaElementi(T dati) {
foreach (T item in dati) {
Console.WriteLine(item.ToString());
}
}
}
```

otterremo un errore di compilazione il quale ci indica che il nostro tipo T potrebbe non essere una collection.Esiste però la possibilità di limitare i tipi <T> attraverso i constraints (vincoli).
Essi ci consentono di passare un numero limitato di tipi e ci aiutano a ridurre gli errori che potrebbero sorgere durante l'uso dei nostri componenti. Un constraint si definisce con la sintassi

```
Where T: vincolo
```

che, applicato al codice di MyClass si traduce in:

```
class MyClass<T> where T: IEnumerable{
public void ElencaElementi(T dati)
{
foreach (T item in dati) {
Console.WriteLine(item.ToString());
   }
      }
}
```

facendo di fatto sparire l'errore di compilazione.
Se provassimo ad istanziare MyClass con il seguente codice:

```
MyClass<int> test = new MyClass<int>();
```

otterremmo un errore di compilazione dato dal fatto che int non implementa l'interfaccia richiesta dal vincolo.
I constraint applicabili agli oggetti generici sono i seguenti:

- struct: il tipo deve essere un value type.
- class: il tipo deve essere un reference type.
- new(): il tipo deve necessariamente avere un costruttore di default senza parametri.

Quando usato con altri constraint, il vincolo new() deve essere l'ultimo della lista, ad esempio:

```
class MyClass<K, V, X>
where K: IEnumerable
where V: struct
```

```
where X: new()
{
//implementazione della classe
}
```

- nome classe base: il tipo deve derivare da una specifica classe
- nome interfaccia: il tipo deve implementare una specifica interfaccia.

E anche possibile specificare più tipi per uno stesso vincolo, separandoli con una virgola, come ad esempio:

```
class MyClass<T> where T : IEnumerable, new() {
```

L'uso appropriato dei vincoli è estremamente comodo durante lo sviluppo, dando la possibilità di limitare il potenziale numero di errori che i generic, per la loro natura, possono introdurre.

6.2.3 Default

Usando i generics, bisogna prestare attenzione al tipo di ritorno da un metodo. Potendo infatti specializzare una nostra classe o un metodo con un qualsiasi tipo di dato, il valore di default del tipo di ritorno potrebbe generare degli errori. Il valore di default di un tipo valore infatti è 0, mentre quello di un tipo di riferimento è null.
Prendiamo ad esempio il seguente codice:

```
class MyDefaultTestClass {
public static T GetDefaultValue<T>() {
//implementazione del metodo
return 0;
 }
 }
}
```

In fase di compilazione, otterremo il seguente errore: "Cannot implicitly convert type 'int' to 'T'" che ci indica l'impossibilità di convertire in int (0) il tipo T. Se provassimo ad usare invece:

```
class MyDefaultTestClass {
public static T GetDefaultValue<T>() {
//implementazione del metodo
return null;
   }
   }
}
```

otterremo il più significativo errore: "Cannot convert null to type parameter 'T' because it could be a value type.
Consider using 'default(T)' instead" che stà ad indicare la possibilità che T, data la sua natura, potrebbe essere un tipo valore, a cui ovviamente non possiamo assegnare null.
La soluzione ce la fornisce la parola chiave default. Il suo scopo è quello di assegnare, al tipo di ritorno, 0 se il type parameter T è un tipo valore o null se è un tipo riferimento.
L'uso della parola chiave default è molto semplice:

```
class MyDefaultTestClass {
public static T GetDefaultValue<T>()
{T result = default(T);
//implementazione del metodo
 return result; }
   }
}
```

Il nostro codice ora compila senza errori e siamo sicuri che il valore di ritorno dal metodo sia coerente con il type parameter usato per richiamarlo.

6.3 COLLECTION GENERICHE

Uno dei campi in cui i Generics sono maggiormente utili, è quello della realizzazione di collections. Il .NET Framework, fornisce già una serie di collections generiche pronte all'uso che si rivelano estremamente comode per far fronte a numerose esigenze. Sono le corrispettive delle collezioni non generiche viste nel capitolo 9. Tutte le collections di questo tipo incluse nel .NET Framework si trovano sotto il namespace System.Collections.Generics.Anche per le collezioni generiche, come per quelle classiche, è possibile definire collection tipizzate mediante l'implementazione di una serie di interfacce, del tutto simili a quelle viste nel capitolo relativo alle collection, ma anche esse, questa volta, generiche.

6.3.1 List<T>

La prima collection che analizzeremo è una semplice lista generica, denominata List<T>:

```
List<string> myList = new List<string>();
myList.Add("Questo");
myList.Add("è");
myList.Add("un");
myList.Add("esempio");
myList.Add("di");
myList.Add("List<T>");
foreach (string testo in myList)
{Console.WriteLine(testo);}
```

Il suo utilizzo è decisamente semplice: nella prima riga, viene istanziata una collection List specificandone il tipo (in questo caso string). Nelle successive, vengono inseriti gli elementi al suo interno che, come è semplice da comprendere, dovranno essere di tipo string.
Un eventuale inserimento di int, ad esempio, causerà un errore in fase di compilazione.

6.3.2 Stack<T>

Nel capitolo 13.1.2, per comprendere meglio le problematiche di type safety risolte dai generics, avevamo usato un oggetto Stack realizzato da noi. Il .NET Framework, come già anticipato il quel capitolo, include un oggetto Stack generico che possiamo direttamente utilizzare

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
foreach (int value in stack) {
Console.WriteLine(value.ToString());
}
stack.Pop();
stack.Pop();
foreach (int value in stack) {
Console.WriteLine(value.ToString());
}
```

Anche qui, nella prima riga, viene istanziato l'oggetto Stack<T> specificandone il tipo (int nel nostro esempio). Questo oggetto è sicuramente più performante del corrispettivo non generico in quanto, come sappiamo, oltre a mantenere la caratteristica di essere type safe, non necessità di continue operazioni di Boxing/Unboxing.

6.3.3 Queue<T>

Parlando di Stack, avevamo introdotto il principio di LIFO (Last In, First Out), vedendolo praticamente negli esempi precedenti. Esiste anche un oggetto chiamato Queue (coda), molto simile allo Stack ma con logica di ingresso ed uscita degli elementi diversa, ovvero FIFO (First In, First Out). A differenza dello Stack infatti, i primi elementi ad entrare, sono i primi ad uscire:

```
Queue<string> queue = new Queue<string>();
```

```
queue.Enqueue("Hello ");
queue.Enqueue("from ");
queue.Enqueue("queue ");
foreach (string value in queue) {
Console.Write(value.ToString());
}
queue.Dequeue();
queue.Dequeue();
foreach (string value in queue) {
Console.WriteLine(value.ToString());
}
```

6.3.4 Dictionary<K, V>

Le collection di tipo Dictionary sono caratterizzate da una coppia di chiave-valore con cui sono memorizzati gli elementi al loro interno. Uno dei punti di forza delle Dictionary è la velocità di accesso ai membri interni in quanto, questo tipo di collection, usa al suo interno una Hashtable. Vediamo un esempio di Dictionary generica:

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();
dictionary.Add("Valore 1", 10);
dictionary.Add("Valore 2", 20);
Console.WriteLine("Valore 2= {0}", dictionary["Valore 2"].ToString());
```

Potendo passare come type parameter un qualunque tipo, possiamo realizzare ad esempio una Dictionary per la gestione dei nostri oggetti. Si pensi ad esempio all'oggetto Product visto nel paragrafo 9.4.3. La relativa Dictionary sarà:

```
Dictionary<string, Product> dictionary =
new Dictionary<string, Product>();
```

che, a differenza della HashTable usata come esempio, non necessita

di operazioni di Boxing/Unboxing ed accetta solo i due tipi definiti, rendendo il nostro codice più sicuro e performante.

6.4 ANONYMOUS METHODS

Un Anonymous Methods è, come dice il termine stesso, un metodo senza firma (quindi anonimo). Questa definizione, nei fatti, si traduce nella possibilità di avere, al posto della firma del metodo usato dal delegato, direttamente il codice che deve essere eseguito. Vediamo subito un esempio pratico paragonandolo ad un classico delegate.
Con il .NET Framework 1.1 eravamo costretti a dichiarare il delegate e definire il metodo da passare come argomento:

```
 delegate void MyOldDelegate();
 static MyOldDelegate GetOldMethod() {
 return new MyOldDelegate(MyOldMethod);
}
 static void MyOldMethod() {
 Console.WriteLine("Questo è un delegato classico");
}
```

Con gli Anonymous Methos invece, al posto di definire il metodo da passare come argomento, possiamo inserire direttamente il codice che deve essere eseguito:

```
 delegate void MyNewDelegate();
static MyNewDelegate GetNewMethod()
 {return delegate()
{
Console.WriteLine("Questo è un Anonymous Method"); };
}
```

Il seguente codice, che fa uso di entrambi i tipi di delegate:

```
static void Main() {
  MyOldDelegate oldDelegateInstance = GetOldMethod();
  MyNewDelegate newDelegateInstance = GetNewMethod();
  oldDelegateInstance();
  newDelegateInstance();
  Console.ReadLine();
}
```

restituirà come output:

```
Questo è un delegato classico
Questo è un Anonymous Method
```

Come già sappiamo, i delegati sono alla base degli eventi. E' dunque scontato che l'uso degli Anonymous Methods è applicabile anche agli eventi. Vediamo ad esempio come potrebbe cambiare la gestione di un evento click su un bottone.
Se prima avremmo dovuto creare un metodo da passare come argomento al gestore dell'evento, oggi possiamo più sinteticamente scrivere:

```
btnTest.Click +=
delegate(object sender, EventArgs e)
{ MessageBox.Show("Click"); };
```

Il codice risultante è più compatto e anche più elegante. Non è sempre possibile usare gli Anonymous Methods al posto dei più classici delegati ma, ove possibile, il risparmio di codice è evidente.

6.5 PARTIAL TYPES

Questa nuova funzionalità consente di suddividere una classe in più file diversi e di ricostruirla al momento della compilazione.

I vantaggi introdotti da questa caratteristica possono non essere immediati. Cercheremo quindi di scoprirli insieme.
Prendiamo ad esempio un'applicazione Windows Forms realizzata usando ancora il framework 1.1 e Visual Studio .NET 2003. Andando a guardare il codice di uno qualsiasi dei form che fanno parte del progetto, ci renderemo conto di quanto codice è stato inserito in automatico da Visual Studio per definire e posizionare tutti gli elementi dell'interfaccia grafica.Elementi indispensabili, fortunatamente nascosti da una region, ma pur sempre presenti.
Se provassimo ad aprire un form creato invece con Visual Studio 2005, ci troveremmo dinanzi una classe pulita, senza il codice che definisce gli elementi che ne costituiscono l'aspetto.
Ma dove sono finiti? Visual Studio 2005 ha creato per noi un nuovo file denominato NomeForm.Designer.cs in cui, come dice il nome stesso, sono presenti tutti gli elementi di cui parlavamo poc'anzi.
Il codice del form in cui andremo a lavorare sarà quindi pulito e più semplice da navigare.Un tipo parziale si definisce usando la parola chiave partial. Il nostro form sarà quindi definito come:

```
partial class Form1 {
    //codice relativo al form }
```

sia nel file NomeForm.cs che nel file NomeForm.Designer.cs .
Sarà poi compito del compilatore recuperare tutti i tipi definiti partial che hanno lo stesso nome e crearne un unico assembly.In questo modo, il nostro progetto può essere più ordinato e più semplice da gestire.

6.6 ITERATORS

L'utilizzo dello statemet foreach, come abbiamo visto nel paragrafo 9.2, presuppone che la collection sui cui si esegue il ciclo implementi l'interfaccia IEnumerator. Finché si stà utilizzando una delle collection

già presenti nel .NET Framework, il problema non si pone in quanto la suddetta interfaccia è già implementata. Ma se stiamo lavorando con una nostra collection, dobbiamo effettuare il lavoro visto nel paragrafo 9.2. Lo scopo degli Iterators è quello di evitare, nei casi più semplici, la necessità di scrivere una classe che implementi la suddetta interfaccia. Vediamolo con un esempio pratico:

```
class ListClass {
public System.Collections.IEnumerable SampleIterator(int start, int end)
{
for (int i = start; i <= end; i++){
yield return i;
    }
  }
}
```

La nostra ListClass implementa un metodo SampleIterator al cui interno, un ciclo for restituisce il valore di i attraverso la nuova parola chiave yield. Ad esempio

```
 public void test() {
 ListClass test = new ListClass();
 foreach (int n in test.SampleIterator(1, 10))
{
 System.Console.WriteLine(n);
  }
}
```

Non abbiamo avuto la necessità di implementare l'interfaccia IEnumerator. Un esempio più concreto è il seguente:

```
 public class SampleCollection {
 public int[] items;
```

```
 public SampleCollection() {
 items = new int[5] { 5, 4, 7, 9, 3 };
 }
 public System.Collections.IEnumerable BuildCollection()
{
 for (int i = 0; i < items.Length; i++)
 {
 yield return items[i];
   }
  }
}
```

Per utilizzare la SampleCollection sarà sufficiente scrivere il seguente codice:

```
class TestCollection
{
 public static void TestSampleCollection()
{
  SampleCollection col = new SampleCollection();
System.Console.WriteLine("I valori nella collection sono:");
foreach (int i in col.BuildCollection())
{
 System.Console.Write(i + " ");
 }
 }
}
```

Anche qui, come evidente, non è stato necessario implementare la IEnumerator con evidente risparmio di codice.

i libri di

IMPARARE C#

Autore: Michele Locuratolo

EDITORE
Edizioni Master S.p.A.
Sede di Milano: Via Ariberto, 24 - 20123 Milano
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

Realizzazione grafica:
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)

Art Director: Paolo Cristiano
Responsabile di progetto: Salvatore Vuono
Coordinatore tecnico: Giancarlo Sicilia
Illustrazioni: Tonino Intieri
Impaginazione elettronica: Lisa Orrico, Salvatore Spina

Servizio Clienti
☎ **Tel. 02 831212 - Fax 02 83121206**
@ **e-mail: servizioabbonati@edmaster.it**

Stampa: Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Giugno 2006